UQAC
Université du Québec
à Chicoutimi

MÉMOIRE

PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

MOHAMED RACEM BOUSSAHA

SURVEILLANCE DES PROPRIÉTÉS DE SÉCURITÉ AVEC BEEPBEEP

JANVIER 2019

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

RÉSUMÉ

La programmation sécurisée consiste à prendre en compte la sécurité à toutes les étapes du développement d'un logiciel informatique ; cela permet d'éviter au maximum les failles et bogues qu'un attaquant peut exploiter afin de porter atteinte au système sur lequel ce logiciel s'exécute (Viega et McGraw2001). Dans ce mémoire, nous montrons comment l'outil de « run-time monitoring » (surveillance au moment de l'exécution) et traitement de flux d'événements BeepBeep, peut être utilisé pour développer un système flexible et extensible qui permet d'analyser le respect des propriétés de sécurité dans les programme Java. L'approche proposée repose sur AspectJ pour générer une trace qui capture le comportement d'un programme pendant qu'il s'exécute. Cette trace est ensuite traitée par BeepBeep, ce dernier permet d'énoncer et de vérifier des spécifications de differents niveaux de complexité. Selon le résultat renvoyé par BeepBeep, AspectJ est capable d'interrompre l'exécution du programme ou prendre une autre mesure. La méthode proposée offre de multiples avantages, notamment la possibilité de concevoir et d'énoncer des propriétés de sécurité définies par l'utilisateur.

Mots-clés : détection au moment de l'exécution, codes malveillants, propriétés de sécurité, BeepBeep, la programmation sécurisée, AspectJ, traçage de codes informatiques, instrumentation.

INTRODUCTION

CONTEXTE

La sécurité informatique, est un ensemble de stratégies mises en place pour gérer les processus, outils, et politiques nécessaires à la prévention, détection, et documentation des menaces aux informations des utilisateurs des systèmes informatiques (Rouse, 2016). De nos jours, la sécurité informatique est devenue une question primordiale qui préoccupe beaucoup de gens. L'adoption accrue de celle-ci dans des secteurs à sensibilité élevée tels que les gouvernements et les banques, a fait que son marché aie gagné rapidement du terrain au fil des années. En effet, pendant les 13 dernières années, le marché de la sécurité informatique a été multiplié environ 35 fois. En 2004, ce marché valait 3,5 milliards de dollars, tandis qu'en 2017 ce nombre s'est accru à plus de 120 milliards de dollars (Morgan, 2017). Malgré les avancées atteintes grâce aux travaux réalisés dans ce domaine, la sécurité de millions de systèmes informatiques reste imparfaite. En effet, on estime que les produits antivirus disponibles dans le commerce ne sont capables de détecter que 45% des nouvelles menaces auxquelles les internautes sont confrontés chaque jour (Yadron, 2014).

Au cours des dernières décennies, l'avancée technologique de la communication numérique a

rendu l'application de la sécurité informatique encore plus difficile qu'elle ne l'était déjà. Auparavant, un système informatique avait au plus quelques dizaines d'utilisateurs, tous membres de la même organisation. Les utilisateurs exécutaient des programmes écrits en interne ou produits par quelques fournisseurs. Aujourd'hui, avec la décentralisation du processus de développement des logiciels, et la pratique du téléchargement des programmes informatiques depuis des sources inconnues (dans le but de les utiliser et les exécuter localement sur des machines personnelles), des logiciels malveillants « malware » sont apparus sur Internet. Ce type de logiciel circulant sur la toile menace la sécurité des utilisateurs du réseau. Dans ce contexte, un logiciel malveillant désigne tout programme ou fichier nuisible à l'utilisateur d'un ordinateur. Les logiciels malveillants comprennent les virus, les vers, ransomware,les chevaux de Troie, ainsi que les logiciels espions (des programmes qui recueillent sans autorisation des informations sur un utilisateur) (Rouse, 2017).

Selon McAfeeLabs, plus de 150 millions nouveaux codes malveillants ont été créés en 2017 (McAfeeLabs, 2017), cela représente une augmentation de 25% par rapport à 2016. Compte tenu de ces chiffres, la détection des codes malveillants est devenue un problème important. Ceci implique la nécessité de trouver des moyens de se protéger des dommages que peuvent causer ces derniers, et garantir l'intégrité des systèmes informatiques.

Pour la prévention des attaques de codes malveillants, la pratique de la programmation sécurisée consiste à développer des logiciels informatiques sûrs de manière à prévenir l'introduction accidentelle de vulnérabilités de sécurité (Viega et McGraw, 2001). En effet, les défauts, les bogues et les failles logiques sont la cause principale des vulnérabilités logicielles couramment exploitées (Viega et McGraw, 2001).

Lors de la conception de systèmes informatiques, les développeurs essaient de prévoir toutes les possibilités d'attaques potentielles. Le système doit répondre à des spécifications qui

permettent de le protéger contre les menaces ; ces spécifications sont appelées propriétés de sécurité. Couvrir toutes les failles possibles est quasiment irréalisable du fait que de nouvelles techniques d'attaques et de nouvelles failles de sécurité apparaissent en continu, ce qui met en échec ce principe (Evans et Twyman, 1999). Par exemple, une propriété présentée dans (Long et al., 2013) spécifie que dans la programmation Java, dans le cas où nous voulons vérifier si deux clés chiffrées sont égales, la méthode par défaut `java.lang.Object.equals()` est incapable de comparer ces clés, parce que la plupart des classes dérivées de la classe `java.security.Key` ne redéfinissent pas cette méthode. Par conséquence, la comparaison peut renvoyer le résultat `false` même si les deux clefs sont égales. Cela peut constituer une faille qu'un attaquant pourrait exploiter pour feindre le système. La solution consiste à utiliser la méthode `equals()` comme premier test, puis comparer les versions déchiffrées des clés pour avoir le bon résultat.

Il y a eu une quantité importante de travaux sur les techniques et outils de détection des codes malveillants. Une enquête menée dans Mathur et Hiranwal (2013) a visée à classifier ces techniques. Dedans les auteurs distinguent deux types d'analyseurs : analyseurs statique et analyseurs dynamiques. Les analyseurs statiques sont capables de détecter les défauts dans le code source sans l'exécuter. En revanche, les analyseurs dynamiques doivent exécuter le code pour détecter les défauts. Ces analyseurs peuvent être basés-signature ou basés-comportement. Dans le premier type, les signatures sont créées en examinant le code malveillant. Le code est analysé et les fonctionnalités sont extraites. Ces fonctionnalités sont utilisées pour construire la signature d'une famille de logiciels malveillants particulière. Les analyseurs basés-comportement tentent de caractériser le comportement d'un programme et permettent de détecter des malwares connus ou inconnus (Mathur et Hiranwal, 2013).

PROBLÉMATIQUE

Étant donné que la plupart des analyseurs existants sont basés sur la signature, il existe toujours un intervalle de temps entre le moment où un code malveillant apparaît pour la première fois et le moment où la signature correspondante qui permet de le détecter est dérivée et distribuée aux utilisateurs. Pour les logiciels malveillants tels que le ver SQL Slammer, tout écart de plus de quelques heures est inacceptable, car un ver bien conçu peut neutraliser Internet en quelques heures (Moore et al., 2003). De ce fait, l'importance de la détection précoce des failles, en plus de permettre aux développeurs d'ajouter de nouvelles propriétés et signatures sans effort considérable ; est un fait bien établi pendant la phase des tests de sécurité dans le processus de développement des logiciels (Lam et al., 2006).

En 2015, Ray et Ligatti abordent la question d'offrir plus de flexibilité à l'utilisateur d'un système informatique en lui rendant possible la spécification de ses besoins en termes de sécurité. La sécurité d'un programme informatique a traditionnellement été modélisée comme un prédicat spécifiant si le programme sous analyse est sécurisé ou non-sécurisé ; un programme est sécurisé seulement s'il respecte toutes les propriétés de sécurité prises en compte par le système de sécurité. Dans le cas contraire, il est considéré comme étant non sécurisé. Autrement dit, au lieu de spécifier si les systèmes sont sécurisés ou pas, ils précisent le niveau de sécurité de ces systèmes (Ray et Ligatti, 2015). De même, lors de la phase de test, dans le processus de développement des logiciels informatiques, les développeurs chargés des tests, devraient avoir la possibilité de choisir les propriétés de sécurité à appliquer pendant l'analyse d'un programme informatique selon leurs besoins. Par exemple, si un développeur de logiciels informatique désire savoir si un programme se connecte au réseau, il n'a pas à analyser toutes les propriétés existantes dans le moniteur ; Cela impliquera, des coûts superflus en argent et temps d'exécution.

PROPOSITION

Selon Leucker et Schallhart (2009), le *« runtime monitoring »* (la surveillance au moment de l'exécution) permet l'analyse de l'exécution d'un système informatique en se basant sur l'extraction des détails de son exécution, et les utilise pour détecter et réagir à un comportement satisfaisant ou violant une certaine propriété au moment de l'exécution. Le *« runtime monitoring »* utilise les ressources partagées du système sur lequel le moniteur est exécuté, cela peut causer une diminution des performances. Chaque événement est extrait et analysé avant sont exécution, cela peut ralentir le fonctionnement du programme (Leucker et Schallhart, 2009).

Plusieurs outils adoptant cette technique ont été développés. Parmi eux, un moniteur extensible et léger appelé BeepBeep a été développé par Hallé (2015). L'outil de monitoring BeepBeep permet la vérification d'une spécification donnée par un programme informatique au fur et à mesure de son exécution. Par exemple, les messages envoyés ou reçus par l'application peuvent être vérifiés par cet outil en se basant sur une spécification donnée en tant que propriété de sécurité (prédéfinis sous forme de contraintes) (Hallé, 2015). Ce moniteur a la spécificité de se présenter comme un outil facile à manipuler : il permet d'ajouter, modifier ou supprimer facilement des spécifications au système surveillé.

QUESTION DE RECHERCHE

Dans notre travail de recherche, nous nous proposons de répondre à la question suivante : est-il possible de développer un modèle flexible et extensible qui permet la détection des violations des propriétés de sécurité dans un programme informatique Java en utilisant l'outil de *« runtime monitoring »* BeepBeep ?

MÉTHODOLOGIE

Dans le but d'implémenter un système de *« runtime monitoring »* des propriétés de sécurité dans les programmes Java, nous commençons par concevoir un modèle comportant des mécanismes appropriés qui permettent l'instrumentation du programme et la détection des violations des propriétés de sécurité.

Pour l'instrumentation, AspectJ, une extension dédiée à la programmation orientée aspect, est utilisée pour insérer du code aux points que nous définissons dans un programme Java, AspectJ se présente comme un outil très utile pour le *«runtime monitoring»*.

Dans une seconde étape, nous définissons un ensemble de propriétés de sécurité susceptibles de menacer un système informatique. Afin de définir ces propriétés nous avons exploré la littérature. L'approche Naccio (Evans et Twyman, 1999) définit un large éventail de propriétés de sécurité, nous en avons sélectionné quelques-unes qui constituent une base initiale des propriétés de sécurité dans notre travail. Nous ajoutons ensuite d'autres propriétés plus complexes non traitées par Naccio afin de démontrer l'extensibilité de notre approche. Des chaînes de processeurs BeepBeep qui permettent la détection des violations de ces propriétés de sécurité au moment de l'exécution seront implémentées par la suite.

Pour la validation, nous implémentons un nouveau moniteur d'exécution qui adopte le modèle proposé et permet la détection des violations des propriétés de sécurité choisies par l'utilisateur dans les programmes Java. Ce moniteur nommé **Stethoscope** permet aussi d'ajouter, modifier et supprimer les signatures associées à ces propriétés.

Comme dernière étape, nous étudions l'impact du fonctionnement du moniteur sur la performance du système surveillé et l'utilisateur du moniteur. Ce dernier ne doit pas nuire à l'exécution du programme sous surveillance et ne doit pas présenter des difficultés à être

manipulé. Nous le testons sur des traces de programmes de différentes longueurs ; allons de 0 à 1 million événements. Nous mesurons ensuite l' « overhead » généré ainsi que la difficulté que présente l'utilisateur à l'utiliser ou étendre la plage de propriétés de sécurité qu'il prend en charge.

CONTRIBUTION

La facilité de manipulation, et l'extensibilité qu'offre BeepBeep, donnent aux utilisateurs du moniteur la capacité d'étendre la plage des propriétés de sécurité prises en charge. L'utilisateur pourra ajouter de nouvelles propriétés, supprimer ou modifier des propriétés déjà existantes, rapidement et sans présenter des efforts considérables.

Cette approche permet aux utilisateurs de spécifier leurs exigences de sécurité en spécifiant les propriétés de sécurité qu'ils désirent analyser. Par le *« runtime monitoring »* on automatisera l'analyse des programmes au moment de l'exécution. Il n'y aura besoin de l'interaction de l'utilisateur que pour spécifier les contraintes de sécurité souhaitées, notre application sélectionnera automatiquement l'événement et l'analysera.

ORGANISATION DU MÉMOIRE

Ce mémoire est organisé en cinq chapitres :

Le chapitre 1, pour une meilleure compréhension du travail effectué, introduit les notions de base de la programmation sécurisée et du *« runtime monitoring »*.

Le deuxième chapitre dresse un état de l'art des techniques de détections des codes malveillants dans les systèmes informatiques. Il permet ainsi d'apprécier l'avancée apportée par l'approche

proposée, au domaine de la programmation sécurisée.

Dans le troisième chapitre, nous abordons plus en profondeur le principe et l'architecture de notre approche. Nous décrivons son fonctionnement, les différentes étapes d'implémentation pour son intégration au code, ainsi que la façon de l'appliquer et l'utiliser pour écrire des spécifications. À partir de ces informations, il sera possible d'intégrer cette solution à notre champ d'intérêt. Ici le fonctionnement du moniteur BeepBeep est formulé ainsi que la méthode d'instrumentation de ce dernier via AspectJ.

Dans le quatrième chapitre, nous décrivons les propriétés qui constituent notre base de propriétés de sécurité. Ensuite, nous expliquons la conception et l'implémentation de celle-ci. Nous commençons par des propriétés simples, puis pour montrer la puissance de notre approche, nous abordons des propriétés plus complexes.

Dans le cinquième chapitre, afin d'expérimenter notre approche, nous proposons Stethoscope, un nouvel outil de détection des violations des propriétés de sécurité dans les programmes Java construit autour de l'outil de traitement d'événement de flux BeepBeep. Par la suite, nous testons ce moniteur sur des programmes téléchargés sur Internet et analysons les résultats obtenus en vue de valider l'efficacité de l'approche.

Enfin, la conclusion résume notre contribution au domaine de la programmation sécurisée et donne nos perspectives et nos pistes de recherche futures.

La majorité des travaux réalisés dans ce mémoire ont été publiés dans l'article suivant :

Mohamed Racem Boussaha, Raphaël Khoury et Sylvain Hallé. "Monitoring of security properties using BeepBeep". Dans : 10th International Symposium on Foundation of Privacy and Security, FPS 2017, Revised Selected Papers , October 23-25, 2017, Nancy, France, p. 160-169.

CHAPITRE 1

NOTIONS DE BASE

Ce premier chapitre fournit des informations nécessaires pour la bonne compréhension du reste du mémoire. La première section, comporte des concepts de base dans le domaine de la programmation sécurisée ainsi qu'une classification des codes malveillants et des failles pouvant menacer la sécurité d'un système informatique. Ensuite, le *« runtime monitoring »* est abordé avec des explications qui permettent d'introduire le lecteur à ce domaine.

1.1 INTRODUCTION À LA PROGRAMMATION SÉCURISÉE

Viega et McGraw, (2001) definit la programmation sécurisée comme suit :

« la programmation sécurisée consiste à prendre en compte la sécurité informatique à tous les moments de la conception, la réalisation et l'utilisation d'un programme informatique. »

La programmation sécurisée vise à eviter au maximum les failles de sécurité, afin d'atteindre cet objectif, les développeurs de logiciels informatiques doivent respecter des propriétés et politiques de sécurité qui permettent de prévenir les attaques de logiciels malveillants. Le passage d'un programme sécurisé à non-sécurisé est souvent sensible ; parce que des actions ou des décisions qui peuvent être non liées à la programmation sécurisée, peuvent affecter la

sécurité d'un système (Viega et McGraw, (2001).

1.1.1 PROPRIÉTÉS DE SÉCURITÉ

Selon Zakinthinos (1996), les propriétés de sécurité spécifient les types de comportements qu'un programme informatique est autorisé à faire, pour qu'un programme soit considéré comme sûr, il faut qu'il satisfasse ces propriétés de sécurité. Une **propriété** de sécurité peut consister en soit une interdiction d'accès à une ressource du système, par exemple interdire à un programme de lire les données internes de l'utilisateur, soit une restriction sur le comportement du programme, par exemple, limiter le nombre d'octets qui peuvent être écrits dans le système (Zakinthinos, 1996).

1.1.2 POLITIQUE DE SÉCURITÉ

selon (Evans et Twyman, 1999), une **politique** de sécurité est un ensemble de propriétés de sécurité et leurs paramètres. Les politiques de sécurité peuvent être combinées pour définir une nouvelle politique. Les politiques de sécurité sont exprimées en termes de ressources abstraites et sont indépendantes de la plateforme (Evans et Twyman, 1999).

1.1.3 MENACE

Comme définit dans (Schiller, 2002), une attaque (ou exploit), correspond à une action entreprise pour nuire à une ressource ; Autrement dit, il s'agit d'un événement potentiel, malveillant ou autre, qui peut nuire à un élément de valeur, tel que les données d'une base de données ou d'un système de fichier, ou une ressource du système (Schiller, 2002).

1.1.4 MALWARE

Un logiciel malveillant (Malware) est un logiciel qui pénètre le système de l'utilisateur et porte menace à ce dernier. De nos jours, les malwares continuent de croître en nombre et en complexité. Après être entré dans le système, le logiciel malveillant recherche les vulnérabilités de ce derniers et les exploite pour mener des activités imprévues sur le système (Goyal et Sharma, 2015). Les codes malveillants peuvent être classifiées en plusieurs types, que nous décrirons brièvement ci-dessous :

Virus

Les virus informatiques sont des programmes informatiques capables de se répliquer et d'infecter un ordinateur. Ils ont été nommés ainsi dû à leur ressemblance avec les virus biologiques en ce qui est de la capacité de se multiplier. De la même façon qu'un virus biologique, ils trouvent un hôte, puis l'infectent et se multiplient. Certains virus et autres programmes malveillants présentent des symptômes visibles pour l'utilisateur, mais beaucoup sont passent inaperçus. Un virus informatique ne peut se propager d'un ordinateur à un autre que lorsque son hôte l'ait transféré sur l'ordinateur cible ; par exemple, si utilisateur l'a envoyé à travers un réseau ou sur Internet, ou l'a porté sur un support amovible tel qu'un CD, un DVD ou un lecteur USB. La transmission de virus peut augmenter en infectant des fichiers sur un système de fichiers réseau ou sur un système de fichiers auquel un autre ordinateur a accès. Les virus sont parfois confondus avec les vers informatiques et les chevaux de Troie, techniquement différents (Khan, 2012).

Après avoir pris contrôle d'une application, un virus peut modifier ses paramètres, supprimer ses données ou même bloquer tout le système. Il n'est pas évident pour un utilisateur de

suivre le problème. Il existe de nombreux types de virus, les virus fonctionnent comme des parasites qui infectent les systèmes. Ces programmes sont sous la forme d'un fichier exécutable possèdent l'extension .exe ou .com qui contrôle un autre programme, lorsque quelqu'un clique sur le fichier .exe dans l'ordinateur, le virus affecte le système, parfois bloque le système entier qui doit être formaté à nouveau (Evans et Twyman, 1999).

Chevaux de Troie

Dans (Goyal et Sharma, 2015) un cheval de Troie (*« Trojan Horse »* en anglais) est défini comme un malware injecté par son concepteur dans une application qui paraît inoffensif ; L'application accomplit une action illégale, mais semble remplir certaines fonctions utiles. Les attaquants utilisent ce malware pour lancer leurs attaques à distance et utilisent les systèmes cibles à leurs propres fins, ils peuvent ainsi acquérir des mots de passe, surveiller ce qui se passe sur le système ou corrompre les fichiers systèmes (Goyal et Sharma, 2015).

Les vers informatiques

Un logiciel qui effectue ses copies en exécutant son propre code, indépendamment de tout autre programme, est appelé ver (*« worms »* en anglais), ils ont pour but de prendre le contrôle du systeme, voler des informations confidentielles de l'utilisateur, soit pour les convertir en «zombies» ou «bots» télécommandés (Barwise, 2010). Il envoie des copies de lui-même à d'autres systèmes du réseau sans l'approbation de l'utilisateur. Ces vers consomment la bande passante du réseau pour l'arrêter, se propagent via des liaisons réseau et visent à infecter la plupart des systèmes informatiques associés au réseau sans exiger l'existence d'un fichier contrairement aux virus (Goyal et Sharma, 2015). En conséquence, il pourrait chiffrer, supprimer des fichiers, ou envoyer des courriers indésirables

Ransomware

Un ransomware est un malware qui exige un paiement en échange d'une fonctionnalité ou données volée, ce terme est construit sur les deux mots rançon et malware. Selon Gazet (2010), la plupart des ransomwares répandus utilisent le chiffrement de fichier comme moyen d'extorsion. Les ransomwares ont été utilisés pour des extorsions massives et ont été largement répandus parmi de nombreux utilisateurs (Gazet, 2010).

1.1.5 VULNÉRABILITÉ

Dans la sécurité informatique, une vulnérabilité est une faiblesse qui permet à un attaquant de réduire l'assurance de l'information d'un système, cette vulnérabilité est l'intersection de trois éléments : la susceptibilité ou le défaut du système, l'accès de l'attaquant au défaut et la capacité de l'attaquant à exploiter le défaut (Hughes et Cybenko, 2014). Par exemple, elle peut consister en un défaut lié à la gestion des mots de passe où l'utilisateur utilise des mots de passe faibles qui pourraient être découverts par force brute (Pauli, 2017). Dans un autre cas, l'utilisateur de l'ordinateur stocke le mot de passe sur l'ordinateur où un programme peut y accéder. Les utilisateurs aussi peuvent réutiliser les mots de passe entre de nombreux programmes et sites Web (Vacca, 2012). Les bogues de programme aussi posent différents types de menaces de sécurité, un attaquant peut délibérément les exploiter où ils peuvent accidentellement causer directement des dommages. Les avis de sécurité enregistrés par CERT regorgent d'exemples de programmes bogués menant à des vulnérabilités de sécurité exploitables (Evans et Twyman, 1999).

1.1.6 CONTRE-MESURE

Une contre-mesure est une action de protection qui contre une menace et atténue un risque. Différents types de menaces exigent des contre-mesures différentes. Cependant, il est impossible de créer un système résistant à toutes les menaces existantes. Dans notre travail nous ignorons les menaces qui ne découlent pas de l'exécution des programmes et nous portons notre attention sur celles liées à l'exécution de programmes candidats à provoquer des vulnérabilités menaçant la sécurité du système utilisateur.

Pour faire face à ces menaces des contre-mesures spécifiques permettant de renforcer le système ont été développées, selon Evans et Twyman (1999) ces contre-mesures se présente sous deux formes :

Les restrictions sur lesquelles les programmes peuvent s'exécuter peuvent être basées sur la confiance (seuls les programmes qui sont signés par quelqu'un de confiance), ou basée sur l'analyse statique qui prouve qu'un programme n'a pas certaines propriétés indésirables (exécuter seulement les programmes qu'après une vérification avec un détecteur de virus, ne contiennent pas de séquences d'instructions correspondant à des virus connus)(Evans et Twyman, 1999).

Les contraintes sur ce que les exécutions peuvent faire peuvent être exprimées comme une propriété. La propriété qui doit être appliquée à une exécution dépend de la confiance que l'utilisateur a dans le programme et de la quantité de connaissances disponibles sur son comportement attendu. Idéalement, toutes les exécutions avec une propriété qui les limite à exactement le comportement jugé acceptable pour ce programme (Evans et Twyman, 1999). Toutefois, cela n'est pas possible car les utilisateurs ne peuvent pas s'attendre à rechercher et encoder les limites du comportement attendu pour chaque programme avant de l'exécuter.

Au lieu de cela, nous devons utiliser des propriétés différentes comme contre-mesures pour différents types de menaces (Vacca, 2012).

1.2 INTRODUCTION AU *«RUNTIME MONITORING»*

Pour expliquer clairement ce qu'est le «runtime monitoring», il est important de préciser la signification de quelques mots ou appellations utilisés dans le domaine du «monitoring».

Pour commencer, selon Leucker et Schallhart « *Un moniteur est un dispositif qui lit une trace finie et qui rapporte un certain verdict* » (Leucker et Schallhart, 2009). C'est l'entité qui va surveiller l'exécution du système. Le moniteur peut soit être directement intégré au système, ou opérer sous la forme d'un programme indépendant. Le verdict indique si une spécification a été violée ou non. Les résultats possibles sont : vrai, faux et non concluant. La trace représente l'exécution du système. C'est à partir d'elle que le moniteur peut obtenir le verdict d'une spécification qui décrit le comportement attendu du système et est exprimée en termes des séquences d'événements qu'il doit produire.

Une spécification ou propriété est une description de ce qu'un programme est censé faire. Les spécifications sont écrites pour comprendre et affiner les applications déjà bien développées. Les spécifications critiques, comme pour la sécurité sont souvent précisée avant le développement des applications. Le respect des spécifications est important pour que les systèmes restent stables. Par exemple, la spécification HasNext expliquée dans (Hallé et al., 2014), spécifie que l'interface Java Iterator nécessite que la méthode `hasNext()` soit appelée et renvoie `true` avant l'appel de la méthode `next()`, si cela ne se produit pas, il est très possible qu'un utilisateur répète la lecture d'une collection. Une autre spécification appelée ReadPrint empêche un événement `print` après l'observation d'un événement `read`.

Certaines propriétés sont très particulières, comme les propriétés *data race* et *deadlock*. Dans (Boyapati et al., 2002) les auteurs spécifient qu'un *data race* se produit lorsque deux threads accèdent simultanément aux mêmes données sans synchronisation, et qu'au moins un des accès est une écriture. Les auteurs ajoutent qu'un *deadlock* se produit lorsqu'il existe un cycle de la forme : $\forall i \in 0..n-1$, $Thread_i$ détient $Lock_i$ et $Thread_i$ attend $Lock_{(i+1)modn}$. Les erreurs de synchronisation dans les programmes multithreads sont parmi les erreurs de programmation les plus difficiles à détecter, reproduire et éliminer, ces propriétés sont généralement souhaitées être satisfaites par tous les systèmes et peuvent être mieux mises en œuvre algorithmiquement (Boyapati et al., 2002).

Le *«runtime monitoring»* est une approche d'analyse de l'exécution d'un système informatique basée sur l'extraction d'informations d'un système en cours d'exécution et son utilisation pour détecter et éventuellement réagir à des comportements observés satisfaisant ou violant certaines propriétés (Leucker et Schallhart, 2009). Dans une recherche conduite dans (Varvaressos, 2014), les auteurs définissent des termes liés au runtime monitoring comme suit :

Le **système monitoré** est le système que nous tentons d'analyser et détecter les anomalies présentes dedans, et ce système, peut consister en un système pour utilisation simple, comme il peut consister en un système à utilisation sensible tel que les banques et les gouvernements. En parallèle avec l'exécution du système monitoré, des **événements** indiquant l'état du système, et les actions effectuées au sein de ce dernier, sont produite, ces événements servent à instrumenter le moniteur (Varvaressos, 2014). La figure 1.1 représente le fonctionnement d'un tel système.

Dans le runtime monitoring, les spécifications de vérification sont généralement exprimées dans des formalismes de prédicats de trace, tels que des machines à états finis, des expressions régulières, des grammaires sans contexte, des expressions de logique temporelle linéaire, etc.,

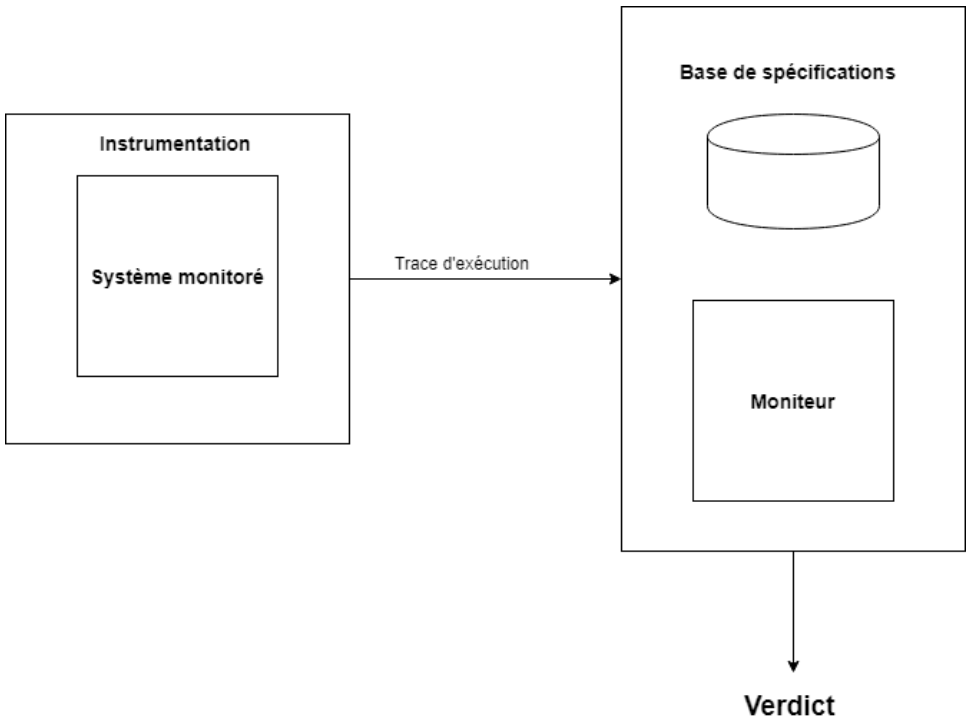


Figure 1.1 – Représentation graphique du fonctionnement d'un moniteur (Varvaressos, 2014)

ou des extensions de celles-ci (Varvaressos, 2014).

Le runtime monitoring nécessite une sorte d'instrumentation qui sonde le système en exécution et rapporte les informations sur ce dernier. L'instrumentation peut être insérée sous la forme de lignes de code dans le programme surveillé, à chaque fois que le programme atteint un certain point dans l'exécution, le code de l'instrumentation s'exécute. Cette instrumentation a un impact considérable sur les performances, et entraîne une augmentation significative de la taille du code (Tran et al., 2008). Plusieurs autres techniques permettent de réaliser l'instrumentation. Parfois, l'instrumentation est assistée par matériel comme avec les points d'arrêt matériels du débogueur. Sur l'architecture X86, un CPU fournit quatre registres de débogage (DR0, DR1, DR2, DR3), chacun contenant l'adresse mémoire à récupérer. L'exécution du système peut être interceptée, en manipulant les registres de débogage (Tian et al., 2017). La prise en charge matérielle de la surveillance et de l'exécution capture les propriétés

souhaitables de l'instrumentation, avec des coûts d'exploitation potentiellement inférieurs en temps d'exécution (Nagarakatte et al., 2010). D'autres approches d'instrumentation existes, comme l'instrumentation basée-minuterie (p. ex., (Metz et al., 2005)), ou l'instrumentation basée-compteurs (p. ex., (Arnold et Ryder, 2001)), mais dans notre mémoire nous nous concentrons sur une technique en particulier qui est **l'instrumentation basée sur la programmation orientée aspect** (Nusayr et Cook, 2009).

La programmation orientée aspect (POA) (Kiczales et al., 1997) est une approche de développement logiciel utilisée pour résoudre les problèmes de la séparation des préoccupations transversales. Dans le développement logiciel, la séparation des préoccupations transversales consiste à organiser et modulariser des parties similaires d'un logiciel, telles que l'authentification, le débogage, la sécurité, la gestion des transactions, la mise en cache, qui couvrent la totalité de l'ensemble des modules logiciels contenus dans les préoccupations principales. L'aspect intègre toutes les fonctionnalités transversales qui ne doivent pas être répliquées aux différents endroits dans le programme source. Cela ne peut pas être résolus efficacement avec la programmation orientée objet (POO), la POA complète donc la programmation orientée objet (Schatten et al.2010).

Un modèle de base POA définit certains indicateurs de pointcuts (point de coupe) (PCD), qui sont des caractéristiques de l'exécution du programme associé à des points de jointures (Joinpoint). Un point de jointure est une étape de l'exécution du programme. Par exemple, il peut consister en l'exécution d'une méthode avec un certain nom ou le traitement d'une exception. La déclaration d'un pointcut comprend deux parties ; une signature comprenant un nom et tous les paramètres, et une expression pointcut qui détermine l'ensemble de Joinpoint concernés par le pointcut (Johnson et al., 2004). L'exemple suivant représente la déclaration d'un pointcut qui correspond à l'exécution de toute méthode principale `main` d'un programme :

```
private pointcut mainMethod () :
execution (public static void main (String[]));
```

Les pointcuts doivent être utiles et pratiques à implémenter à un programmeur d'aspect dans le système POA. En consequence, les pointcuts sont généralement des points du programme où l'insertion d'instructions n'est pas trop difficile ; Par exemple, les appels de méthodes sont très souvent utilisés comme l'un des indicateurs fondamentaux de points. Le système POA le plus populaire, AspectJ, implémente le paradigme POA pour Java. Ses concepteurs de pointcuts incluent les appels de méthode, les exécutions de méthode, les accès aux champs d'objets, les exceptions et quelques autres (Nusayr et Cook, 2009).

Des Advices sont associés à chaque expression de pointcut et s'exécutent à touts points de jointure correspondant au pointcut. Un advice est une action prise par un aspect à un point de jointure particulier. Un advice peut être de différents types, le tableau suivant représente les différents types d'advices disponibles dans AspectJ.

Type de Advice	Fonction
Before	Exécuté avant un point de jonction
After returning	Exécuter après la fin normale sans exceptions d'un Joinpoint
After throwing	Exécuté si une méthode quitte en lançant une exception
After (finally)	Exécuter sans prendre en compte le moyen par lequel un Joinpoint se termine (retour normal ou exceptionnel)
Around	Advice qui entoure un point de jointure tel qu'une invocation de méthode

Tableau 1.1 – Taxonomie des Advices dans la programmation orientée aspect

Comme exemple, nous présentons un aspect de minuterie qui mesure la durée de l'exécution d'une méthode cible dans un programme. Nous incluons cet exemple car il est suffisamment

simple pour comprendre facilement le paradigme de la programmation orientée aspect :

Afin de réaliser cela, nous implémentons deux classes : La première classe `Test`, est une classe Java classique représentée dans le listage 1.1, cette classe imprime un message sur la console de l'utilisateur en appelant la méthode `Test.cible()`. La deuxième est une classe de type spécial Aspect, c'est ici que sont définis les pointcuts, advices, ainsi que le comportement qu'on désire ajouter aux points de jointure désirés dans le programme principal. Le listage 1.2 représente cet aspect qui a pour fonction de mesurer la durée de l'exécution de la méthode `Test.cible()`.

Listing 1.1 – Code source du programme "Test"

```
1 public class Test {
2 public static void cible (){
3 System.out.println("Execution␣...");
4 }
5 public static void main(String[] args) {
6 cible();
7 }
8 }
```

La sortie du programme avant l'ajout de l'aspect est la suivante :

```
Execution...
```

Listing 1.2 – Code source de l'aspect "AspectMinuteur"

```
1 public aspect AspectMinuteur {
2
3 pointcut pt() : execution (void Test.cible());
4
5 long startTime;
6
```

```
7  // S'execute avant le pointcut
8  before() : pt()
9  {
10 startTime = System.currentTimeMillis();
11 System.out.println("Before␣the␣execution\nStart␣time:␣"+
12 startTime);
13 }
14
15 // S'execute apres le pointcut
16 after() : pt()
17 {
18 long endTime = System.nanoTime();
19 long duration = (endTime - startTime);
20 System.out.println("Duree␣de␣l'execution:␣"+duration);
21 }
22 }
```

À l'exécution, l'aspect est tissé au code de l'application aux différents points de jointure. Le tissage est l'étape de la programmation orientée aspect qui a pour but d'insérer des bouts de codes appelés greffons dans le code d'une application ; tisser un greffon dans une méthode revient à redéfinir la méthode comme la composition de son corps avec le greffon (Baker et Hsieh, 2002). Autrement dit, selon l'advice, avant ou après chaque appelle de méthode concerné par le pointcut déclaré dans l'aspect, le greffon du code est exécuté. La nouvelle sortie du programme `Test` est la suivante :

```
Before the execution
Start time: 1545164380491
Execution ...
Execution time: 1
```

La recherche conduite dans (Nusayr et Cook, 2009), les auteurs affirment que la création de l'instrumentation nécessaire et la rendre efficace sont des tâches complexes et technologiquement difficiles, impliquant souvent la création d'outils d'instrumentation spécialisés à la main. Ils avancent aussi que les coûts d'instrumentation vont de l'effet à peine perceptible à l'extrême pénibilité en termes d'impact sur les performances des applications.

CHAPITRE 2

REVUE DE LITTÉRATURE

Ce chapitre est consacré à l'étude des différentes méthodologies utilisées dans le domaine de la détection des failles et codes malveillants dans les programme informatiques. Ceci est réalisé afin d'identifier les points faibles des autres méthodes et justifier l'intérêt d'introduire la nôtre.

Nous commençons par définir les critères d'analyse qui permettent l'évaluation des performances de notre méthode par rapport aux autres. Ensuite, les méthodes utilisées pour la détection des failles et codes malveillants dans les programme informatiques sont étudiées, il est important de prendre en compte le fonctionnement du milieu. Finalement nous appliquons nos critères d'analyse à ces dernières, ce qui permet d'identifier si elles les respectent ou pas.

2.1 CRITÈRES D'ANALYSE

Pour être utile, un système de sécurité de code est conçu pour fournir un code robuste, autrement dit un système de sécurité doit garantir la **sécurité** et **l'efficacité** à l'utilisateur tout en assurant trois objectives clefs dans notre approche : **flexibilité, facilité d'utilisation, extensibilité**.

2.1.1 SÉCURITÉ

La sécurité est une propriété essentielle de tout système de sécurité du code informatique. Un système de sécurité du code sécurisé, doit détecter les enfreintes aux propriétés de sécurité sélectionnées même en présence d'attaquants motivés et bien informés qui tentent de feindre ou compromettre le système.

Chaque système a des vulnérabilités, mais les systèmes de sécurité devraient s'efforcer d'éliminer les vulnérabilités connues et de réduire la probabilité que les attaquants puissent trouver et exploiter des vulnérabilités inconnues (Evans, 2000).

2.1.2 EFFICACITÉ

Dans un système de sécurité l'efficacité est primordiale, Le système de sécurité devrait optimiser la consommation des ressources utilisées dans la production du résultat, la consommation des ressources devrait être évidente à l'utilisateur seulement dans les situations alarmantes ou un programme est sur le point de commettre une enfreinte à la sécurité ; cela signifie que le monitoring en arrière-plan du programme exécuté doit avoir des effets minimes au déroulement d'un programme en effort et temps d'exécution (Evans, 2000). Cela ne devrait certainement pas avoir d'impact sur les programmes qui ne sont pas limités par une politique de sécurité (Evans et Twyman, 1999). Les utilisateurs n'utiliseront pas un système de sécurité du code s'il introduit des coûts généraux inacceptables dans les délais de téléchargement ou d'exécution du code, et les administrateurs des systèmes ne le déploieront pas si cela nécessite des conditions inacceptables de configuration, de traitement ou de stockage (Evans et Twyman, 1999). L'efficacité peut être mesurée à partir du rapport entre les résultats obtenus et les ressources utilisées.

2.1.3 FLEXIBILITÉ

La flexibilité permet à l'administrateur système de régler la sensibilité de la détection des logiciels malveillants. Par exemple, le système peut devenir plus ou moins sensible aux logiciels malveillants en changeant un paramètre.

Avoir un système qui permet aux utilisateurs de spécifier différentes politiques de sécurité selon le niveau de confiance accordé aux programmes exécutés ajoute de la flexibilité, sans limiter le fonctionnement du système quand le code exécuté est valide ou de source sûre.

2.1.4 FACILITÉ D'UTILISATION

Selon les auteurs dans (Evans, 2000), un système de sécurité de code n'est utile que s'il peut appliquer des politiques qui imposent des contraintes utiles au comportement du programme. En outre, il doit y avoir un moyen de définir ces politiques. Si le système est trop difficile ou lourd lorsqu'il s'agit de définir des stratégies, seules des politiques prédéfinies seront disponibles pour les utilisateurs.

2.1.5 EXTENSIBILITÉ

En raison de la vitesse d'apparition des codes malveillants, un système de sécurité doit permettre l'extensibilité et le renforcement des politiques de sécurité en ajoutant de nouvelles contraintes au système de sécurité ou en imposant celles qui sont déjà existantes (Evans, 2000). Il est important que le système de sécurité soit extensible pour que de nouvelles propriétés de sécurité puisse être ajoutées avec un effort raisonnable. Bien que certains travaux devront inévitablement être nécessaires pour définir des propriétés, la conception du système de sécurité doit maximiser la réutilisabilité. Autrement dit une fois que l'architecture de la plate-forme soit comprise, ce que doit être fait pour ajouter de nouvelles fonctions a celle la,

doit être clair.

Dans la section suivante, nous présentons une discussion sur les techniques de détection des logiciels malveillants, en mettant l'accent sur celles qui sont à la base de la recherche présentée dans ce travail.

2.2 ANALYSE DE CODE MALVEILLANT

Il y a plusieurs approches de détection de codes malveillants. Ces techniques peuvent être classifiées en deux catégories, les techniques basées sur la signature et les techniques basée sur le comportement.

La technique antivirus la plus utilisée est la détection basée sur la signature (Kirda et al., 2006). Une signature est une séquence d'octets pouvant être utilisée pour identifier des logiciels malveillants spécifiques. Les logiciels antivirus basés sur les signatures doivent conserver une base des signatures des logiciels malveillants connus et cette base doit être mise à jour fréquemment lorsque de nouvelles menaces sont découvertes. La détection basée sur la signature est simple, relativement rapide et efficace, par-contre elle nécessite une base de données de signatures à jour, les logiciels malveillants non présents dans cette base de données ne seront pas détectés.

La détection basée sur le comportement se concentre sur les actions effectuées par le logiciel malveillant lors de l'exécution. Dans les systèmes basés sur les comportements, le comportement des logiciels malveillants et des fichiers bénins est analysé au cours d'une phase d'apprentissage. Ensuite, lors d'une phase de test (surveillance), un programme est classé comme logiciel malveillant ou bénin, sur la base de modèles dérivés lors de la phase d'apprentissage. La surveillance du comportement, une étape importante dans le processus d'analyse, est utilisée pour observer les relations malveillantes par rapport au système et est

réalisée en utilisant une instrumentation dynamique sur le système cible (Chess et West, 2007).

Ces techniques de détection de logiciels malveillants utilisent soit l'analyse statique soit dynamique, représentant deux méthodologies importantes pouvant être utilisées pour analyser les logiciels malveillants (Malin et al., 2008).

L'analyse statique du logiciel est effectuée sans exécuter le programme. Des exemples d'informations que nous pouvons obtenir à partir de l'analyse statique incluent des séquences d'opcodes (extraites en désassemblant le fichier binaire), ou des graphes de flux de contrôle. Ces ensembles de fonctionnalités peuvent être utilisés individuellement ou en combinaison pour la détection de logiciels malveillants (Brand et al., 2011).

En revanche, l'analyse dynamique exécute le logiciel malveillant et observe l'interaction de celui-ci avec le système. Cette analyse peut nous fournir des informations tels que : les écritures mémoires, les appels API, les appels systèmes, les traces d'instructions, les modifications du registre, etc.

Dans les points qui suivent, nous citerons et analyserons quelques approches de détection statique et dynamique proposées dans l'analyse des logiciels malveillants qui sont en rapport avec la solution que nous proposons.

2.2.1 LA DÉTECTION DYNAMIQUE BASÉE SUR LA SIGNATURE

Dans (Ahmed et al., 2009) les auteurs proposent un schéma de détection des programmes malveillants qui extrait des fonctionnalités statistiques à partir des informations spatio-temporelles disponibles dans les appels d'API de l'exécution des systèmes d'exploitation Windows. Ils présentent un outil de *« runtime monitoring »* qui extrait des caractéristiques statistiques basées sur des informations spatio-temporelles dans les journaux d'appels d'API. L'information

spatiale est constituée des arguments et des valeurs de retour des appels API, tandis que les informations temporelles sont la séquence des appels API. Chaque appel d'API a un nom unique, un ensemble d'arguments et sa valeur de retour.

La figure 2.1, représente l'architecture du schéma proposé. L'outil comprend deux modules : le premier applique des algorithmes d'apprentissage automatique aux données disponibles pour développer un modèle de formation, et le deuxième module extrait les caractéristiques spatio-temporelles au moment de l'exécution et les compare au modèle de formation pour classer un processus en cours comme bénin ou malveillant.

Les informations temporelles sont modélisées à l'aide d'une chaîne de Markov (Cover et Thomas, 1991) à temps discret d'ordre n, avec k états, chaque état correspondant à un appel API particulier. L'ordre d'une chaîne de Markov indique dans quelle mesure les états passés déterminent l'état actuel, c'est-à-dire combien de retards doivent être examinés lors de l'analyse des ordres supérieurs. Au cours de la phase d'apprentissage, deux chaînes de Markov distinctes sont créées pour les traces bénignes et malveillantes, ce qui entraîne des probabilités de transition différentes pour chaque chaîne. Les transitions moins « discriminantes» sont élaguées en utilisant une mesure informationnelle / théorique appelée gain d'information (IG)(Ahmed et al., 2009). Les transitions avec les valeurs les plus élevées d'IG sont sélectionnées comme entités booléennes. Les caractéristiques spatiaux-temporelles extraites sont ensuite données en tant qu'entrée des algorithmes d'apprentissage automatique pour la classification. Les algorithmes utilisés sont : «Instance Based Learner» (IBk), «Decision Tree» (J48), «Naıve Bayes» (NB), «Inductive Rule Learner» (RIPPER), et «Support Vector Machine» (SMO) expliqués dans (Cover et Thomas, 1991).

Pour l'instrumentation, les auteurs ont utilisé un traceur d'appels d'API commercial qui utilise le «Kernel-Mode Hook» qui est une technique de Hooking qui permet d'enregistrer

les journaux des processus en cours sur Microsoft Windows. Par exemple, supposons que nous souhaitons vérifier les autorisations d'interaction des processus avant de leur permettre de tenter toute activité, le code qui gère ces appels de fonction, événements ou messages interceptés est appelé un Hook.

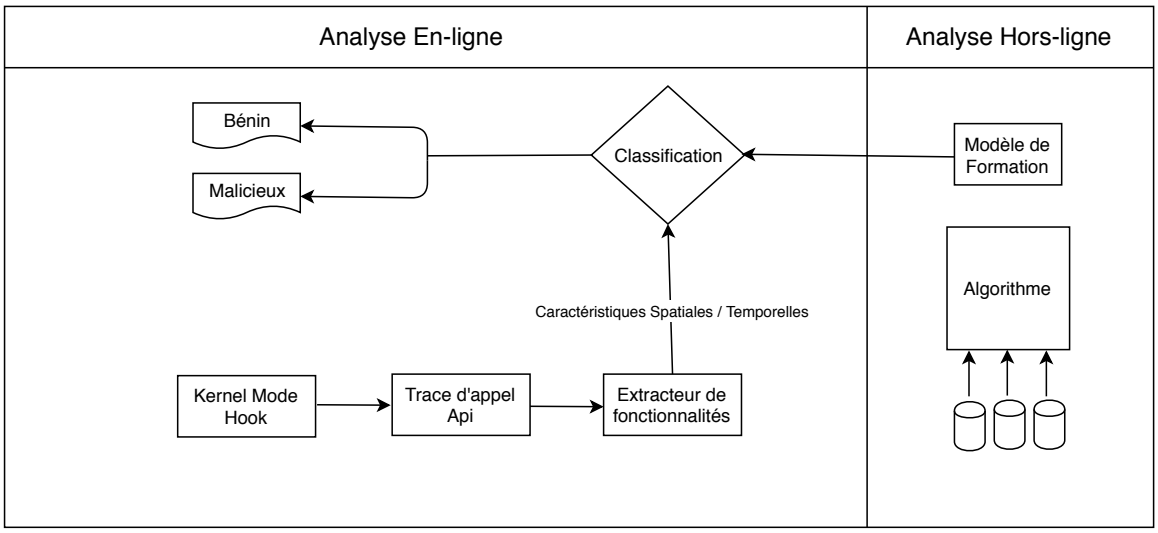


Figure 2.1 – Diagramme de l'outil de détection de malware proposé (Ahmed et al., 2009)

Cette recherche vise à prouver que l'ensemble des caractéristiques spatio-temporelles combinées augmente la précision de la détection par rapport à l'ensemble des caractéristiques spatiales ou temporelles autonomes. Selon les auteurs, ce système atteint la précision de détection de 0.97 en surveillant uniquement les appels d'API provenant de catégories de gestion de la mémoire et d'Entré / Sortie de fichier.

2.2.2 LA DÉTECTION DYNAMIQUE BASÉE SUR LE COMPORTEMENT

Dans Hamann et Mantel (2018) les auteurs présentent **CliSeAuDroid**, un mécanisme d'application dynamique des stratégies de sécurité définies par l'utilisateur pour les applications Android. CliSeAuDroid peut appliquer des stratégies locales et des stratégies distribuées sur le même appareil ou sur plusieurs appareils. L'architecture de CliSeAuDroid comprend quatre

composants : l'intercepteur, le coordinateur, la politique locale et le responsable de l'application. Le composant intercepteur est chargé de surveiller l'application cible et de communiquer les événements du programme interceptés au coordinateur. Le composant coordinateur, vérifie auprès composant de stratégie locale si le comportement du programme observé est conforme aux exigences de sécurité. Au cas où le comportement observé violerait l'exigence de sécurité, une contre-mesure appropriée est déterminée et communiquée au responsable de l'application. Le composant de mise en œuvre implémente cette contre-mesure au point d'interaction avec l'application cible (Hamann et Mantel, 2018).

L'implémentation de CliSeAuDroid repose sur CliSeAu (Gay et al., 2014), une implémentation du framework d'automates de services pour les programmes Java. CliSeAu est conçu de manière modulaire, cette conception modulaire permet de réutiliser de grandes parties de la base de code existante (Hamann et Mantel, 2018).

Le processus d'instrumentation de CliSeAuDroid agit sur le fichier de package d'application (APK) de l'application cible. Les APK sont des fichiers de conteneur, y compris les fichiers binaires de l'application, ainsi que les dépendances nécessaires et les ressources de l'application. L'instrumentation fonctionne sur trois artefacts d'entrée : le fichier APK de l'application cible, une instanciation de CliSeAuDroid pour l'application cible et une spécification de point en coupe des points de programme relatifs à la sécurité.

CliSeAuDroid prend en charge l'application de stratégies de sécurité distribuées en communiquant avec d'autres applications dans le même système. Ce système fait référence à des stratégies qui prennent en compte tous les nœuds d'un système distribué et qui nécessitent une connaissance globale de l'état du système.

L'évaluation expérimentale entreprise par les auteurs, indique que l'overhead ajouté par cette application est faible et qu'il ne peut pas être reconnu par l'utilisateur. Lors de l'application

distribuées des propriétés de sécurité, l'overhead est supérieur, mais reste inférieure à 1 seconde (Hamann et Mantel, 2018).

E-ACSL (Kirchner et al., 2015) est un outil de runtime montoring de la sécurité des programmes C dans Frama-C (Kirchner et al., 2015), un framework dédié à l'analyse du code source des programmes écrits en langage C. E-ACSL utilise un programme C annoté avec des spécifications de sécurité écrites dans le langage de spécification E-ACSL et génère un nouveau programme C intégrant un moniteur généré à partir de la spécification E-ACSL. Au moment de l'exécution, le programme surveillé se comporte de la même manière que le programme d'origine si les propriétés formelles sont satisfaites ou abandonne son exécution si une propriété est violée. Frama-C fournit plusieurs plug-ins pour générer des annotations E-ACSL à partir de spécifications de haut niveau. La plupart des utilisations de E-ACSL ne nécessitent pas d'annotation manuelle des programmes. La vérification des propriétés à l'exécution avec E-ACSL se fait automatiquement (Kirchner et al., 2015).

Le langage de spécification E-ACSL est décrit formellement dans le manuel de référence (Signoles, 2018). Ce langage est un langage de spécification comportementale dont les termes sont des expressions C étendues avec des mots-clés et des fonctions intégrées permettant de gérer les spécificités du C (Kirchner et al., 2015).

L'outil de vérification d'exécution E-ACSL est implémenté en tant que plug-in de Frama-C, Frama-C est en chargée du pré-traitement, de l'analyse, du typage et de la fourniture d'un arbre de syntaxe (AS) représentant le code C en entrée. E-ACSL génère à son tour un nouvel AS qui représente le programme C en sortie intégrant des moniteurs générés à partir d'annotations E-ACSL. Deux analyses statiques sont exécutées avant la génération du code en sortie (Kirchner et al., 2015). Une analyse identifie les instructions de type goto qui vont à l'extérieur des blocs afin d'insérer correctement le code nécessaire lors de la sortie d'un bloc.tandis que l'autre

recherche une sur approximation des instructions à surveiller pour vérifier les propriétés de la mémoire (Jakobsson et al., 2016). Deux autres analyses sont effectuées lors de la phase de génération de code. Tout d'abord, le système traite des entiers mathématiques E-ACSL : pour tout terme E-ACSL, il calcule le type C le plus petit pouvant le coder en toute sécurité. Si le terme ne peut pas être représenté en toute sécurité par un type C, le code généré utilise la bibliothèque arithmétique à multiples précisions Gmp 2 () pour le coder. Enfin, le plug-in RTE de Frama-C est utilisé pour empêcher les comportements non définis dans le code généré (Kirchner et al., 2015). La figure 2.2 représente un tel processus.

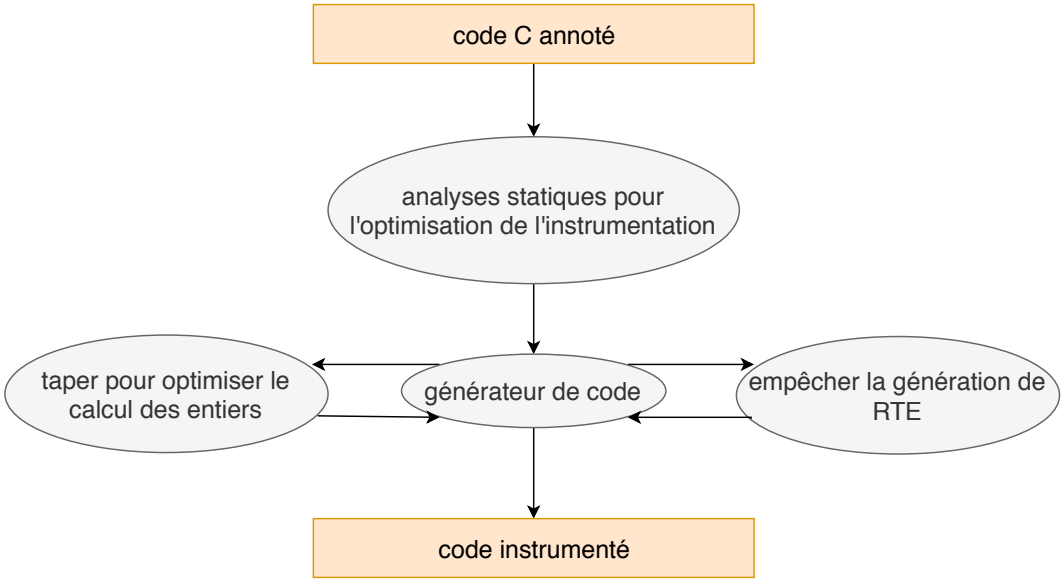


Figure 2.2 – Fonctionnement E-ACSL (Kirchner et al., 2015)

L'analyse dynamique est une technique de détection visant à évaluer les logiciels malveillants en exécutant l'application dans un environnement réel. L'avantage principal de cette technique est qu'elle détecte le chargement de code dynamiquement et enregistre le comportement de l'application pendant l'exécution. Cette technique permet de détecter de nouvelles attaques, mais des attaques déjà définies peuvent lui échapper. Par contre cette méthode génère un « *overhead* » par la détection au moment de l'exécution des contraintes de sécurité ce qui

diminue l'efficacité du système (Guilfanov, 2001). En termes de flexibilité elle ne permet pas à l'utilisateur de spécifier ces exigences par rapport au niveau de sécurité.

2.2.3 LA DÉTECTION STATIQUE BASÉE SUR LA SIGNATURE

Evans et Twyman (1999), présente Naccio, une architecture indépendante de la plate-forme pour la sécurité du code conçue pour offrir une flexibilité supérieure. Naccio peut définir et appliquer des stratégies qui imposent des contraintes arbitraires sur les manipulations de ressources, ainsi que des stratégies modifiant la façon dont un programme manipule les ressources. Cependant, les contraintes ne peuvent ni définir ni imposer de propriétés ou de stratégies dépendantes des propriétés structurelles du code.

Naccio prend un programme et une politique de sécurité et produit un programme qui se comporte comme le programme d'origine, sauf qu'il est garanti de respecter la politique de sécurité. Afin de parvenir à cela, les ressources sont décrites, et ces descriptions de ressources sont utilisées pour définir les politiques de sécurité. Dans cette solution les stratégies de sécurité sont définies à un niveau plus abstrait, ainsi un outil qui génère les Wrappers nécessaires pour appliquer une stratégie sur une plate-forme particulière est fourni. Un Wrappers est une sous-routine dans une bibliothèque de logiciels ou un programme informatique dont le but principal est d'appeler une deuxième sous-routine ou un appel système avec peu ou pas de calcul supplémentaire (Peasley, 1998). Les politiques de sécurité sont exprimées en termes de manipulation de ressources abstraites et une plate-forme en fonction de l'impact de ses appels systèmes sur ces ressources est caractérisée.

La figure 2.3 montre l'architecture du système Naccio. Il est divisé en un générateur de politique et un transformateur d'application. Un auteur de règles exécute le générateur de règles pour produire ce que le transformateur d'application utilise pour appliquer la stratégie sur

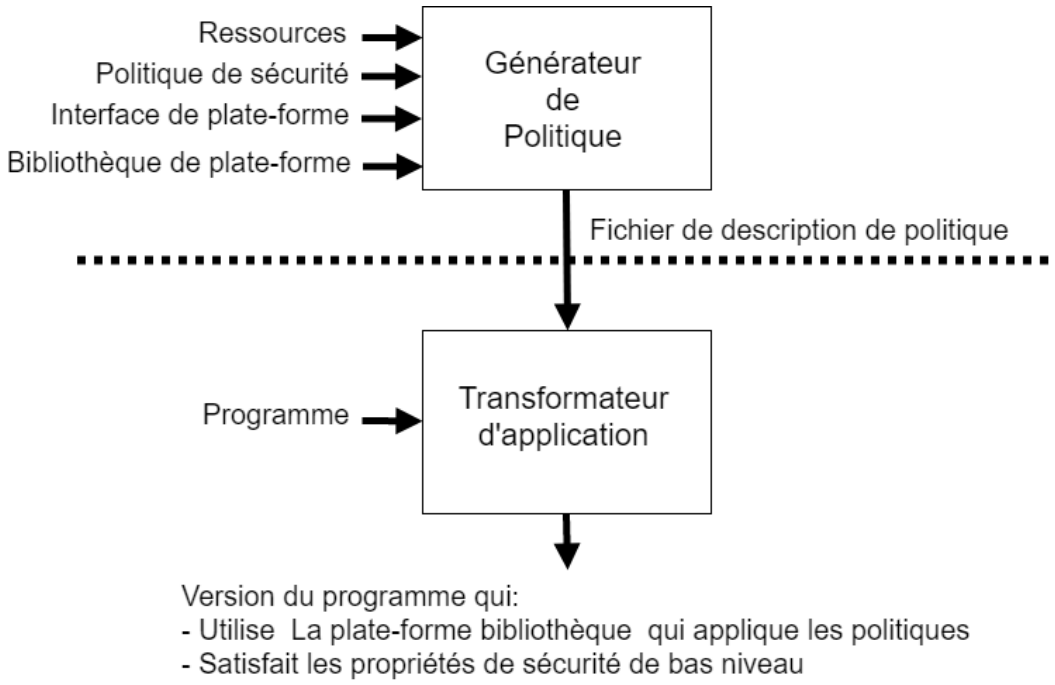


Figure 2.3 – Architecture Naccio (Evans et Twyman, 1999)

un programme particulier. Étant donné que la génération de règles est une tâche relativement peu fréquente, les auteurs compensent le temps d'exécution du générateur de règles pour accélérer la transformation des applications et réduire la surcharge d'exécution associée aux contrôles de sécurité. Une fois qu'une stratégie a été générée, elle peut être réutilisée pour chaque application.

Le projet Naccio (Evans et Twyman, 1999) fournit une bibliothèque de propriétés de sécurité Java qui sont appliquées à l'exécution. Chaque stratégie remplace certaines classes de la « Java Virtual Machine » (JVM) pour permettre l'application, et la JVM doit être modifiée pour garantir que la classe correcte (spécifique à la stratégie de sécurité) est utilisée.

Une politique de sécurité est décrite dans cette approche en répertoriant les propriétés de sécurité et leurs paramètres. Par exemple la stratégie de sécurité LimitWrite qui instancie

deux propriétés de sécurité : NoOverwrite, qui interdit le remplacement ou la modification du contenu d'un fichier existant, et LimitBytesWritten qui limite le nombre de données pouvant être écrites dans des fichiers.

2.2.4 LA DÉTECTION STATIQUE BASÉE SUR LE COMPORTEMENT

Dans la détection statique basée sur le comportement, les caractéristiques de la structure de fichier du programme sous inspection sont utilisées pour détecter les code malveillants. Un avantage majeur de la détection statique basée sur le comportement est que son utilisation peut permettre de détecter les logiciels malveillants sans devoir autoriser l'exécution du logiciel malveillant sur le système hôte.

En 2001, Bergeron et al. ont proposé une architecture de détection statique qui permet la détection des programmes malveillants en analysant le comportement du programme en question. Cette méthode permet la détection de programmes malveillants inconnus. Cela est réalisé en désassemblant le programme en utilisant IDA32 (Guilfanov, 2001). Le programme en assembleur est ensuite analysé pour produire l'arbre de syntaxe. Ensuite le prototype construit le graphe de contrôle-flux pour chaque sous-routine en plaçant des blocs de base sur une liste, puis en convertissant cette liste en un graphique. À l'étape de l'analyse, chaque fois qu'une instruction marquant la fin d'un bloc de base est satisfaite, le bloc de base est construit et les instructions de début et de fin sont stockées dans la liste d'instructions de ce sous-programme. Le graphe de contrôle-flux est ensuite construit.

La figure 2.4 illustre le graphe de contrôle de flux d'un programme WINIPX.EXE, l'examen de ce fichier démontre que les informations sont envoyées sur le réseau et que les informations envoyées proviennent des fichiers du disque (Bergeron et al., 2001).

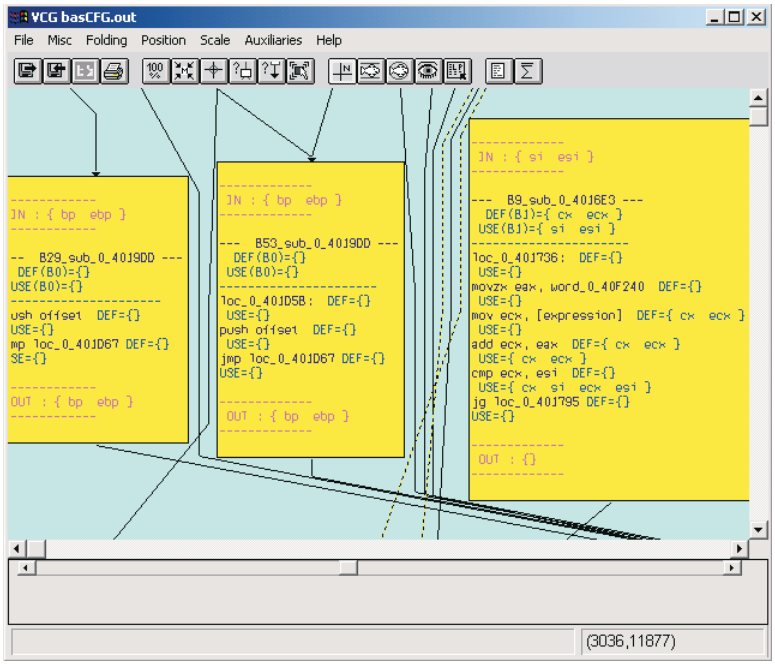


Figure 2.4 – Graphe de flux de contrôle (Bergeron et al., 2001)

Dans cette approche, la politique de sécurité est représentée en automate, les transitions de ces automates sont étiquetées avec les actions que le système peut effectuer. Un des états est désigné comme mauvais état, une entrée à cet état est considérée comme violation à la politique de sécurité. La figure 2.5 représente un automate de sécurité qui renforce une politique de sécurité qui empêche un programme d'envoyer des données sur le réseau après avoir lu un fichier du système. Dans cet automate, τ désigne toute action autre que les actions pertinentes OuvrirFichier, LireFichier et Envoyer, alors que λ désigne toute action.

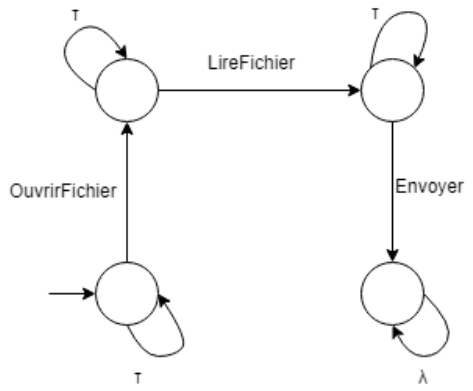


Figure 2.5 – Automate de sécurité (Bergeron et al., 2001)

La politique de sécurité est saisie manuellement sous la forme d'une liste de transitions. Enfin, la politique de sécurité est comparée au graphe de flux de contrôle en utilisant une variante de l'algorithme d'Emerson (Vardi et Wolper, 1986).

En termes de sécurité, cette technique détecte les comportements qui échappent au modèle de comportement prédéfini ou accepté et permet de détecter de nouvelles attaques, tandis qu'elle peut manquer des attaques connues. En matière d'efficacité cette méthode permet d'avoir de bons résultats sans pour autant provoquer un *« overhead »* des performances du fait qu'elle est statique et n'a aucun effet sur l'exécution du programme analysé. On ne prend pas en compte ici la flexibilité par la possibilité de la spécification des propriétés de sécurité à analyser par l'utilisateur.

2.3 CRITIQUE DES SOLUTIONS EXISTANTES

Aucune des études précédentes ne permet à l'utilisateur de spécifier par lui-même les propriétés de sécurité qu'il désire appliquer, ce qui diminue la flexibilité offerte a ce dernier. Nous allons aborder ce problème dans notre étude et implémenter un système qui permet une telle manœuvre.

Les principales contributions de ce travail sont les suivantes :

— Nous capturons des informations détaillées sur le comportement des logiciels Java en exécution. Pour cela nous utilisons la programmation orientée aspect, qui permet d'anticiper les actions malveillantes en extrayant les informations sur les appels de méthodes avant qu'elles soient exécutées et de les arrêter dans le cas ou elles représentent une menace au système.

— Nous avons développé un outil de monitoring flexible qui permet a l'utilisateur d'ajouter, spécifier, modifier des propriétés de sécurité sans présenter des efforts considérables. Cet outil dédié aux développeurs de programmes Java est capable de détecter efficacement les enfreintes des propriétés de sécurité par un programme au moment de l'exécution.

— Nous présentons des preuves expérimentales qui démontrent que notre approche est faisable et utilisable dans la pratique.

L'approche proposée dans ce travail de recherche est précise et peut être appliqué à un code d'origine inconnue et permet à l'utilisateur de personnaliser facilement la politique de sécurité selon ses besoins. En effet, comme nous allons le voir dans les sections suivantes, elle peut être utilisée non seulement pour appliquer une grande variété de politiques de sécurité mais aussi pour assurer le respect des contraintes d'utilisation des ressources ou générer des rapports de diagnostic sur le comportement du programme.

CHAPITRE 3

ARCHITECTURE DE LA SOLUTION

L'architecture système présentée dans ce travail permet de tracer l'exécution d'un programme et de définir des propriétés de sécurité et de vérifier le respect de ces dernières au moment de l'exécution. Conceptuellement, à partir d'un programme et une description d'une propriété de sécurité, un verdict est produit indiquant si le programme respecte cette propriété de sécurité ou pas. Nous pouvons même aller au-delà de ce verdict et fournir des informations plus complexes et plus informatives sur l'exécution du programme surveillé, ce point sera abordé dans le chapitre 4. Cette architecture inclut un traceur utilisant AspectJ qui permet de produire la trace d'exécution des programmes Java, et également un moniteur BeepBeep pour la vérification du respect des propriétés de sécurité en analysant les programmes au moment de l'exécution. Ce chapitre détaille l'architecture de cette approche.

3.1 VUE GLOBALE

De façon générale, la solution consiste à exprimer les propriétés de sécurité à un niveau plus abstrait et de fournir une architecture qui permet de tracer l'exécution d'un programme et de vérifier le respect de ces propriétés au moment de son exécution. Les stratégies de sécurité sont définies en associant la vérification du code à des manipulations de ressources abstraites et exprimant les propriétés en chaînes d'actions élémentaires connectées par des

flux d'événements. La figure 3.1 montre l'architecture du système.

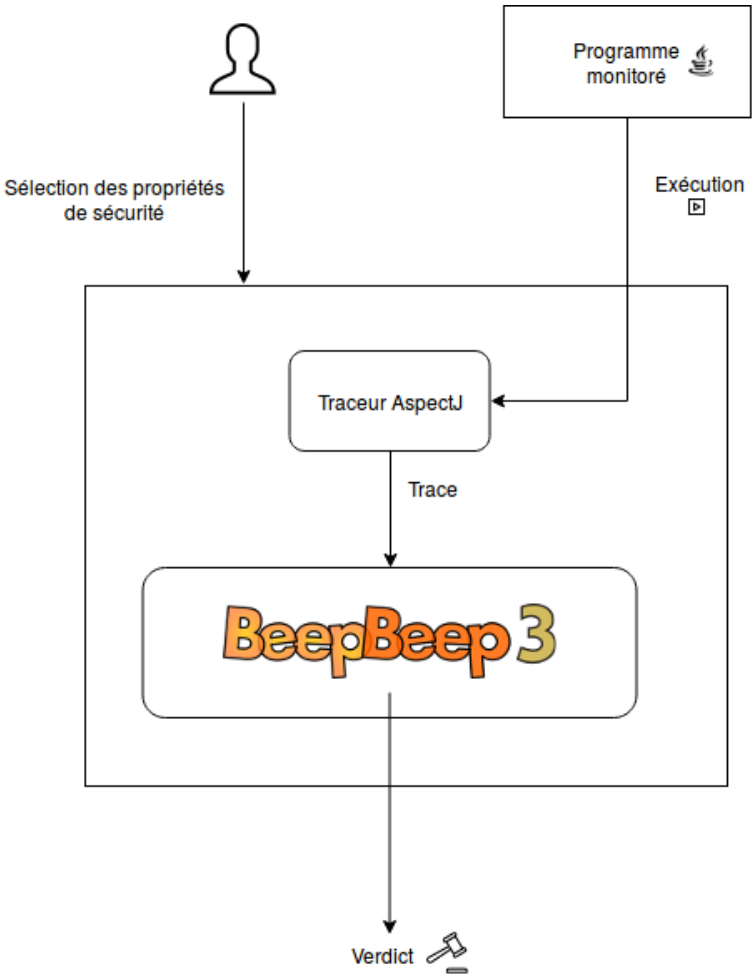


Figure 3.1 – Architecture de l'approche

Le système est divisé en trois parties, le moniteur, le traceur et le programme cible.

Le traceur produit la trace d'exécution du programme cible au fur et à mesure de son exécution. Ligne par ligne, le moniteur reçoit la trace en entrée et l'analyse en fonction des propriétés de sécurité choisies par l'utilisateur. Le moniteur produit un verdict sur le respect de la propriété de sécurité suite à la suite de cette analyse.

Supposons que nous souhaitons appliquer une propriété qui limite le nombre total d'octets qu'un programme peut écrire dans un système. Une implémentation devra maintenir le nombre total d'octets écrits jusqu'au moment présent dans l'exécution. Nous devons vérifier que la limite ne sera pas dépassée avant chaque opération qui écrit dans un fichier. Une façon d'appliquer une telle propriété consisterait à conserver l'état nécessaire et effectuer la vérification requise. Cela nécessiterait la vérification de chaque appel de méthode effectué par le programme cible. Si cette dernière correspond à une méthode qui écrit sur le système le nombre d'octets écrits est extrait et stocké, et cette valeur est contrôlée pour garantir qu'elle ne dépasse pas la limite fixée. Pour appliquer cette propriété, nous devrions analyser l'exécution des programmes cibles. Pour cela notre approche implémente une chaîne d'opérations atomiques qui dans ce cas sera composée de la vérification de la signature de chaque ligne de trace ; si cette signature correspond à une méthode qui écrit dans le système, le nombre d'octets écrits est incrémenté. À la fin la vérification si la limite a été dépassée ou pas, si c'est le cas l'infraction sera signalée à l'utilisateur.

3.2 INSTRUMENTATION DE L'EXÉCUTION

Dans cette section la partie instrumentation à l'aide de la programmation orientée aspect (POA) de notre approche est expliquée.

Dans notre approche nous utilisons AspectJ pour observer l'exécution interne d'un logiciel orienté objet afin d'instrumentaliser notre moniteur. L'instrumentation basée aspect facilite l'observation des détails de l'exécution interne.

Pour ce faire nous avons implémenté un aspect `AspectTracer` dans lequel Pointcuts, Joinpoints, et Advices sont conçus pour faciliter l'extraction des données internes.

Notre technique utilise trois types d'`Advices`. `After`, `Before` et `Around` exepliqués aupara-

vant de la méthode d'entrée/sortie, et `Execution` ou `Call` d'une méthode. Les fonctions de l'aspect sont conçues pour avoir un aperçu de l'exécution du système.

3.2.1 ASPECT DE JOURNALISATION

L'aspect `AspectTracer` permet de faciliter et d'observer l'exécution interne du logiciel pendant le test et de générer un rapport de sortie. Il rassemble les informations d'une application et les stocke dans un fichier journal. La journalisation utilisant l'AOP encouragent la réutilisation du code de journalisation, ce qui constitue un avantage.

L'aspect `AspectTracer` a été développé dans AspectJ. Les fonctions d'aspect sont conçues de manière à faciliter l'observation des détails d'exécution des logiciels. Des pointcuts ont été définis dans l'aspect afin de capturer touts les emplacements dans un programme cible. Ces points sont définis et conçus dans l'aspect pour capturer tous les types de points de jointure dans un programme. Les pointcuts sont également modifiés pour limiter la journalisation des points de jointure se produisant dans des points de jointure ou méthodes non concernée la 3.1 donne un aperçu du code.

```
1 pointcut traceMethods():
2     execution (* *.*(..)) &&
3     call (* *.*(..)) &&
4     !within(TracingAspect) &&
5     !within(SecurityConstraints) &&
6     !within(StethoUI) &&
7     !within(ca.uqac..*)
8 }
```

Listing 3.1 – Aperçu du Pointcut de AspectTracer

Des Advice sont définis pour donner des instructions aux pointcuts. Par exemple, thisJoinPoint est défini dans un Advice pour obtenir des informations sur la classe qui invoque une

méthode et pour obtenir la signature et les arguments de la méthode. Il nous permet de suivre les appels de fonction et les exécutions de l'application. L'utilisation de pointcuts de type `execution (* *.*(..))` et `call (* *.*(..))` spécifie l'utilisation de la signature de la méthode pour l'exécution ou les appels, respectivement, de toute méthode sur une instance de classe. L'utilisation de `!Within(AspectTracer)` éxclut la capture des appels de méthodes et l'exécution ne sont pas dans `AspectTracer`. Cela limite la capture des points de jointure. Les lignes qui succedent dans le code effectuent la même fonction pour respectivement, les classes responsables d'analyser la trace, l'interface de l'outil de monitoring, et les fonctions internes du moniteur BeepBeep.

Les méthodes sont définies à l'intérieur de l'aspect de journalisation, appelées avec des advice et utilisées avec des joinpoints spécifiques pour faciliter l'observation des détails de l'exécution.

`AspectTracer` doit être tissé de manière externe avec un logiciel orienté objet, sans affecter le code du programme. Étant générique il ne nécessite pas de changements dans le corps de l'aspect lorsqu'il est utilisé avec une nouvelle application. Le code d'aspect est tissé avec le code du programme et est compilé à l'aide du compilateur AspectJ (ajc). Le compilateur AspectJ est basé sur un compilateur open-source créé dans le cadre de l'environnement de développement intégré Eclipse (IDE) (Meetei et al., 2011). Il peut fonctionner avec d'autres IDE tels que NetBeans et Emacs JDEE, ainsi qu'avec des outils en ligne de commande. Le compilateur AspectJ prend en charge le mécanisme de tissage. Le tisseur accepte le code source Java et le code source d'AspectJ pour créer un code exécutable tissé, prêt à être exécuté.

3.2.2 RAPPORT DE SORTIE

Notre traceur redirige la sortie de l'exécution du programme vers un fichier journal. Les détails d'exécution du programme recueillis lors de l'exécution sont stockés dans ce fichier journal. Par la suite la sortie stockée dans le fichier journal est analysée par le moniteur pour obtenir les détails d'exécution souhaités du programme et faire des calculs là-dessus.

Dans le fichier journal, on trouve les détails de l'exécution comme les méthodes déclarées ainsi que les détails de la liaison dynamique tels que d'où elles sont appelées. Il stocke également des informations tels que le nom de la classe, le nom/type de la méthode, le nom/type du constructeur, les emplacements source, le type d'arguments, le type de signature et d'autres détails requis lors de l'analyse. En analysant et en filtrant le fichier journal, l'observation des détails du logiciel est possible. La figure 3.2 montre la trace d'exécution du programme `"HelloWorld"`listé dans 1.1.

```
call // void // test.main // [Ljava.lang.String; // [Ljava.lang.String;@3e3abc88 // NAN // 0 // NAN // 303563356
call // void // java.io.PrintStream.println // java.lang.String // HelloWorld // NAN // 1 // 245257410 // 1826771953
output // NAN // 1826771953
output // NAN // 303563356
```

Figure 3.2 – La trace d'exécution du programme HelloWorld

Le tableau 3.2 donne la signification de chaque champ selon son index dans la ligne de la trace.

Dans la trace générée on trouve des détails sur l'exécution du programme surveillé. Chaque événement est identifié par un identifiant qui correspond au code de hachage du point de jointure. `AspectTracer` permet aussi d'avoir la profondeur d'un appel de méthode. Supposons qu'une méthode fait appel à une autre méthode au sein de son code source, cette variable permet d'affecter un niveau d'appel différent à chacun des deux appel.

`AspectTracer` trace trois types d'évènements, appel d'une méthode, retour d'une méthode,

Type	Index	Signification
Appel d'une méthode	1	Type de l'événement
	2	Type de retour
	3	Signature de la méthode
	4	Types des paramètres
	5	Valeurs des paramètres
	6	Octet écrits ou lus
	7	Niveau de l'appel
	8	Hash de l'objet appelant
	9	Type du retour
	10	Id de l'événement
Retour d'une méthode	1	Type de l'événement
	2	Valeur retournée
	3	Id de l'événement
Appel de méthode à clef	11	Index de la clef

Tableau 3.1 – Description de la trace

et appel de méthode à clef. Ce dernier type représente des appels de méthodes ou un des paramètres est une clef qui doit être immutable (inchangeable), cette variante nous sera utile dans la suite du travail.

3.3 LE MONITEUR BEEPBEEP

BeepBeep est un outil de « Complex Event Processing (CEP) » (traitement d'événements complexe) développé au Laboratoire d'informatique Formelle (Hallé, 2016) disponible sous licence open source. Il peut effectuer des manipulations complexes sur de grands flux de données. Ce système composé de techniques basées sur le concept d'événements complexes, c'est-à-dire sur des événements qui n'ont pu se produire que parce que d'autres événements se sont produits auparavant. Les événements sont créés de manière ordonnée, propagés en flux et doivent être traités, agrégés et filtrés au moment de l'exécution (Hallé, 2016).

Soit un ensemble arbitraire d'éléments T. Une trace d'événements de type T est une séquence

$e = e_1, e_2...$ où $e_i \in T$ pour tous i. L'ensemble de toutes les traces de type T sera noté T. Dans ce qui suit, l'événement sera en double frappe (par exemple, T, U, ...) et peut se référer à n'importe quel ensemble d'éléments. Les types d'événements peuvent être aussi simples que des caractères uniques ou des nombres, ou aussi complexes que des matrices, des documents XML ou toute autre structure de données définie par l'utilisateur.

BeepBeep a été choisi pour sa facilité d'utilisation et sa capacité à composer ensemble des unités de traitement qui utilisent un large éventail de langages. Une partie de la contribution du travail est de montrer l'énonciation des propriétés de sécurité pilotées par les données des programmes en termes d'un petit nombre de processeurs BeepBeep. Un avantage de l'approche considérée est la facilité avec laquelle la propriété de sécurité souhaitée peut être énoncée.

3.3.1 FONCTIONNEMENT DE BEEPBEEP

En interne, BeepBeep décompose la tâche de traitement de données souhaitée en un certain nombre de processeurs élémentaires, chacun prenant en entrée un (ou plusieurs) flux d'événements, et à son tour, produisant un ou plusieurs flux d'événements. Ces processeurs sont enchaînés ensemble avec la sortie d'un (ou de plusieurs) processeurs acheminés à l'entrée de la suivante, de telle sorte que l'alimentation du flux d'entrée de BeepBeep à travers cette chaîne produit le calcul souhaité. Un manuel complet comportant le fonctionnement detaillé du moniteur BeepBeep est disponible sur son site Web.

Pratiquement tout le traitement des traces d'événements peut être réalisé grâce à l'action d'un processeur, ou une combinaison de plusieurs processeurs connectés l'un à l'autre formant une chaîne de processeurs.

Pour comprendre mieux la notion de chaînes de processeurs et avoir une vision à un niveau

d'abstraction plus haut, nous devons apercevoir les processeurs comme des boites connectées l'une à l'autre par des tuyaux. Chaque boîte correspond à une action produisant un flux d'événements. Les tuyaux servent à transiter les événements entre les processeurs, certains de ces tuyaux sont utilisés pour mener les événements à un processeur (Inputpipe) tandis que d'autres sont utilisés pour collecter les événements produits par un processeur (Outputpipe).

Il y a plusieurs types de processeurs selon le type de calcul qu'ils exécutent sur les événements. Un processeur peut prendre un ou plusieurs flux d'évènements en entrée et produire à sa sortie un ou plusieurs flux d'évènements, le nombre d'entrées est désigné par le terme arité d'entrée ainsi que le nombre de sorties qui est appelé arité de sortie. Le tableau 3.2 liste les types des différents processeurs de base de BeepBeep ainsi que leurs fonctions.

Ainsi de la même abstraction nous représentons dans ce mémoire les chaînes de processeurs par des boites connectées l'une à l'autre par des tuyaux représentant l'Outputpipe et l'Inputpipe des éléments de la chaîne la couleur du pipe permet d'identifier son type. Les pictogrammes à l'intérieur des boites servent à spécifier le type du processeur. La figure 3.3 représente une chaîne de processeurs simple qui permet de faire la somme de deux nombres ; le code qui permet de réaliser cela est montré dans le listing 3.2.

```
1  QueueSource source1 = new QueueSource();
2  source1.setEvents(2, 7, 1, 8, 3);
3  QueueSource source2 = new QueueSource();
4  source2.setEvents(3, 1, 4, 1, 6);
5  Adder add = new Adder();
6  Connector.connect(source1, 0, add, 0);
7  Connector.connect(source2, 0, add, 1);
8  Pullable p = add.getPullableOutput();
9  for (int i = 0; i < 5; i++) {
10  float x = (Float) p.pull();
11  System.out.println("The␣event␣is:␣" + x);
12 }
```

Listing 3.2 – Aperçu du code source d'une chaîne de processeurs

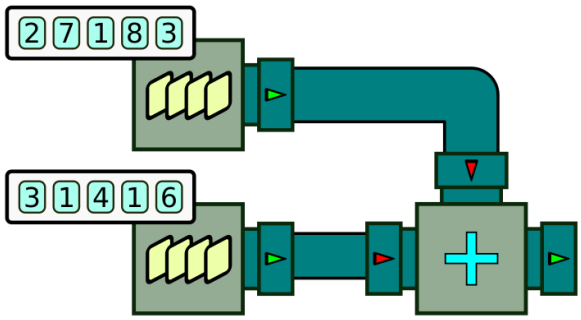


Figure 3.3 – Exemple d'une chaîne de processeurs BeepBeep (Hallé, 2018)

Cette chaîne est composée de trois processeurs, deux processeurs de type QueueSource qui extraient la trace et approvisionnent le reste de la chaîne en événements. Dans ce cas, les événements sont des nombres. Le troisième processeur reçoit ces nombres et les additionne, L'arité d'entrée de ce dernier est égale à deux, puisque chaque paire de nombres acheminée à son entrée est additionnée. La figure 3.4 représente la sortie de la chaîne de processeurs de cet exemple.

```
The event is: 5.0
The event is: 8.0
The event is: 5.0
The event is: 9.0
```

Figure 3.4 – Le résultat d'exécution d'une chaîne de processeurs (Hallé, 2018)

Pour extraire des résultats en sortie d'un processeur, il existe deux modes d'interaction. Le premier mode, utilisé dans cet exemple, s'appelle Pull. Ici, un objet spécial appelé Pullable est instancié. Un `Pullable` agit comme un itérateur sur la trace de sortie. Il fournit une méthode, appelée `pull()`, et chaque appel à cette méthode demande au processeur correspondant de produire un événement de sortie supplémentaire.

La huitième instruction du code prend soin d'obtenir une instance de `Pullable` correspondant à la sortie de `QueueSource` en utilisant une méthode appelée `getPullableOutput()`.

La seconde manière s'appelle Push et fonctionne en sens inverse du Pull. En mode Push,

plutôt que d'interroger les événements de la sortie d'un processeur, nous transmettons des événements à l'entrée d'un processeur. Cela a pour effet de déclencher le calcul du processeur et de pousser les événements vers la sortie.

3.3.2 ÉTENDRE BEEPBEEP

BeepBeep a été conçu pour être facilement extensible ; il contient un noyau de processeurs et fonctions intégrés. Le reste de ses fonctionnalités est implémenté avec des processeurs personnalisés, regroupés dans des packages appelés palettes. Dans le cas ou aucun des processeurs ou fonctions des palettes existantes ne soit approprié à un problème particulier, BeepBeep permet de créer facilement des objets en étendant simplement certaines des classes fournies dans la bibliothèque principale. Dans (Hallé, 2018) deux méthodes permettent de réaliser cela.

Création de processeurs personnalisés

BeepBeep permet de créer de nouveaux processeurs personnalisés, qui peuvent ensuite être composés avec des processeurs existants. Cette tâche est simple, par le fait qu'elle consiste à étendre la classe SingleProcessor. L'utilisateur doit écrire le traitement qui doit avoir lieu comme fonction du processeur, c'est-à-dire quel événement de sortie à produire en fonction d'un événement d'entrée. Dans le code suivant, nous créons un processeur qui permet d'inverser une chaîne de caractéres :

```
1  public class StringInvers extends SingleProcessor {
2
3   public StringInvers() {
4    super(1, 1);
5   }
6
7   public boolean compute(Object[] inputs, Queue < Object[] > outputs) {
8    String res = new StringBuilder(inputs[0]).reverse().toString();
9    return outputs.add(new Object[] {
10    res
```

```
11    });
12   }
13
14   @Override
15   public Processor duplicate() {
16    return new StringInvers();
17   }
18  }
```

Listing 3.3 – Exemple de création d'un processeurs personnalisés

La méthode compute définit le traitement effectué par le processeur. Autrement dit, c'est ici que la fonctionnalité du processeur est implémentée. La spécification de l'arité d'entrée et de sortie du processeur se fait par l'appel de `super()` dans le constructeur de ce dernier, le premier argument est l'arité d'entrée et le second argument est l'arité de sortie. Ces deux valeurs sont égales à un dans notre exemple. La méthode `duplicate()` retourne une nouvelle instance de la classe StringInvers.

Création de fonctions personnalisées

La deuxieme maniére d'étendre BeepBeep consiste à créer une nouvelle fonction personnalisée. Une fonction personnalisée est tout objet qui hérite de la classe de base `Function`. Il existe deux façons principales de créer de nouvelles fonctions :

Étendre la classe `Function` ou l'un de ses descendants tels que `UnaryFunction` ou `BinaryFunction`. Dans un tel cas, la méthode la plus générique consiste à étendre directement la classe abstraite Function et à implémenter toutes les méthodes requises. Il y en a six :

- La méthode `evaluate` est responsable du calcul réel, elle reçoit un tableau d'arguments d'entrée, et écrit dans un tableau d'arguments de sortie.
- Les méthodes `getInputArity` et `getOutputArity` déclarent l'Inputarité l'Outputarité de la fonction, respectivement. Ils doivent renvoyer un seul nombre entier.

— La méthode `getInputTypesFor` est utilisée pour spécifier le type des arguments d'entrée de la fonction. La méthode `getOutputTypeFor` fait la même chose pour les valeurs de sortie de la fonction.

— La méthode `duplicate` doit renvoyer une nouvelle instance de la fonction.

Comme exemple, le code suivant comporte une fonction qui permet d'inverser une chaîne en caractères :

```
1  public class CustomInvers extends Function {
2   @Override
3   public int getInputArity() {
4    return 1;
5   }
6
7   @Override
8   public int getOutputArity() {
9    return 1;
10  }
11
12  @Override
13  public Function duplicate() {
14   return new CustomDouble();
15  }
16
17  public void evaluate(Object[] inputs, Object[] outputs) {
18   outputs[0] = new StringBuilder(inputs[0]).reverse().toString();
19  }
20
21  public void getInputTypesFor(Set < Class << ? >> s, int i) {
22   if (i == 0)
23    s.add(String.class);
24  }
25
26  public Class << ? > getOutputTypeFor(int i) {
27   if (i == 0)
28    return String.class;
29   return null;
30  }
31 }
```

Listing 3.4 – Exemple de création d'une fonction personnalisée

La deuxième façon consiste à **étendre** `FunctionTree`. Un FunctionTree est une arborescence dont les nœuds sont des fonctions simples qui sont composées de façon à créer des fonctions plus complexes. Le constructeur de cette classe doit appeler le constructeur de FunctionTree et créer l'arbre de fonctions approprié. Par exemple, une arborescence qui calcule la fonction $f(x,y,z) = x+y+z$ est construite comme suit :

```
1 public class CustomFunctionTree extends FunctionTree {
2  public CustomFunctionTree() {
3   super(Numbers.addition,
4    new FunctionTree(Numbers.addition,
5     StreamVariable.X, StreamVariable.Y),
6    StreamVariable.Z);
7  }
8 }
```

Listing 3.5 – Chaîne de processeurs simple

Ici un nouvel objet FunctionTree est instancié. Le premier argument est la fonction à la racine de l'arbre qui correspond a l'opérateur d'addition dans notre cas. Le côté gauche de l'addition est lui-même un `FunctionTree`. L'opérateur de cet arbre est aussi l'addition, suivi de ses deux arguments. Puisque nous voulons ajouter les événements provenant des premier et deuxième flux, ces arguments sont deux objets `StreamVariable.` Par convention, `StreamVariable.X` correspond au numéro de flux en entrée 0, alors que `StreamVariable.Y` correspond au flux en entrée numéro 1. Enfin, le côté droit de la multiplication est `StreamVariable.Z`. `CustomFunctionTree` peut être utilisé ensuite n'importe où comme suit :

```
1
2 Function f = new CustomFunctionTree();
3 ApplyFunction af = new ApplyFunction(f);
4 ...
```

3.3.3 SYNCHRONISATION DES ÉVÉNEMENTS

Dans le cas ou un processeur posséde plus qu'une seule entrée, le traitement de celles-ci est synchronisé. Autrement dit le traitement est réalisé seulement si un événement peut être consommé de chaque trace en entrée. Cela simplifie la définition et l'implémentation de processeurs par le fait de ne pas prendre en compte la synchronisation des événements pendant leurs conception. Le résultat en sortie est sensible à l'ordre dont les événements arrivent ou le temps de traitement nécessaire au processeur afin de produire des résultats.
Les processeurs comportent des tampons pour stocker les événements en entrée jusqu'à ce qu'un événement en sortie puisse être produit. Ces tampons sont gérés par les processeurs. Cela est réalisé implicitement. Dans BeepBeep, cela se fait avec la classe abstraite `SingleProcessor`, les descendants de cette classe doivent simplement implémenter une méthode nommée `compute()`, qui est appelée uniquement lorsqu'un événement est prêt à être consommé à chaque entrée (Hallé et al., 2016).

La figure 3.5 représente un exemple de chaîne de processeurs assez similaire à celle représentée par la figure 3.3, sauf qu'ici la première source de la file d'attente a été remplacée par une source de file d'attente lente, qui attend cinq secondes avant de sortir un événement. Ceci est représenté par le pictogramme horloge dans le premier Queuesource.

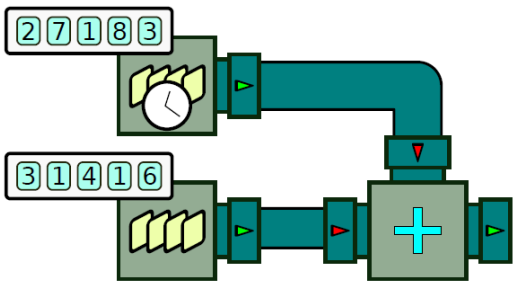


Figure 3.5 – Chaîne de processeurs BeepBeep (Hallé, 2018)

La sortie de ce programme est identique au premier, cependant une ligne est imprimée toutes les cinq secondes. Lorsque la sortie du processeur **add** est sollicitée, le processeur vérifie si les événements à son entrée peuvent être consommées. Il demande à la fois à **source1** et à **source2** un nouvel événement, **source2** répond immédiatement, mais **source1** prend cinq secondes avant de produire un événement. En attendant, **add** ne peut rien faire d'autre qu'attendre. L'ensemble du processus se répète à chaque appel ultérieur à `pull()`. L'appel ne demande qu'un seul événement à la fois pour chaque source, c'est-à-dire qu'il ne continue pas à tirer sur **source2** pendant qu'il attend une réponse de **source1** (Hallé, 2018).

Dans notre travail nous projetons de définir les propriétés de sécurité en tant que chaînes de processeurs dans le moniteur. En outre, le moniteur de programme doit assurer que les propriétés de sécurité du code nécessaires pour empêcher les programmes malveillants de menacer la sécurité du système. Une fois que le programme analysé, il pourra être modifié pour faire en sorte que la propriété de sécurité soit respecté. Afin d'expliquer l'expressivité de notre approche nous avons pris une multitude d'exemples de propriétés de sécurité qui constituent notre base de propriétés et nous les avons modélisées par des chaînes de processeurs BeepBeep qui permettent de détecter les violations à ces propriétés dans le chapitre suivant.

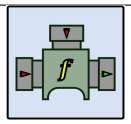
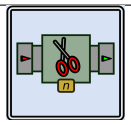
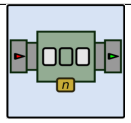
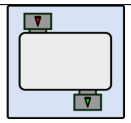
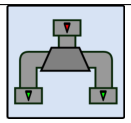
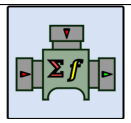
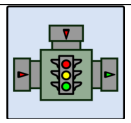
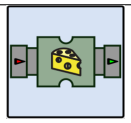
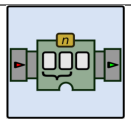
Processeur	Fonction	Représentation
Fonction	Spécifie une fonction à appliquer aux flux	
Trim	Supprime un nombre fixe d'événements depuis le début d'un flux	
Decimate	Rejette les événements d'un flux d'entrée à intervalles réguliers	
Group	Regroupe plusieurs processeurs dans un seul	
Fork	Multiplie le flux en entré en plusieurs	
Cumulitative	Cumule les flux selon une fonction donnée	
Filter	Rejette les événements d'un flux d'entrée d'une manière complètement arbitraire	
Slice	Sépare un flux en plusieurs sous-flux selon une fonction donnée	
Window	Effectuer un calcul sur une fenêtre du flux d'entrée	

Tableau 3.2 – Typologie des processeurs de base de BeepBeep

CHAPITRE 4

PROPRIÉTÉS DE SÉCURITÉ

Dans ce chapitre nous avons commencé par répliquer plusieurs des propriétés de sécurité présentes dans la bibliothèque de Naccio (Evans et Twyman, 1999). La plupart d'entre elles sont des propriétés de sécurité qui peuvent être appliquées avec aussi peu que deux processeurs BeepBeep. Ces propriétés incluent :

- `NoExec`
- `NoJavaClassLoader`
- `NoNetReceiveing`
- `NoNetSending`
- `NoPrinting`
- `NoReadingFiles`
- `NoListingFiles`
- `LimitBytesWritten`
- `LimitBytesRead`

Les sept premières de ces propriétés arrêtent simplement l'exécution en rencontrant un appel de méthode interdit spécifique.

Les quatre dernières propriétés citées sont plus impliquées. LimitBytesWritten et LimitBytesRead limitent le nombre total d'octets écrits (ou lus) dans des fichiers ou sur le réseau. LimitCreatedFiles et LimitObservedFile limitent le nombre de fichiers pouvant être créés (ou lus).

Autre que les propriétés présentées dans Naccio, d'autres propriétés ont été implémentées, à citer :

- `NoSendAfterReading`
- `IsAKey`
- `AsRaedAsWrite`
- `SafeLock`

Puisque la sortie d'un processeur peut être de n'importe quel format, BeepBeep peut également fournir des informations de profilage sur l'exécution en cours, telles que la profondeur maximale, minimale et moyenne des piles, le nombre d'objets créés pour chaque type d'objet, etc. Par conséquent BeepBeep nous permet d'énoncer des propriétés plus complexes qui relient les valeurs présentes dans différentes parties de la trace les unes aux autres. Les deux propriétés, `CallSequenceProfiling` et `LimitBytesWrittenGraph` expliquées dans la suite, ont été implémentées afin de prouver cela.

4.1 PROPRIÉTÉ DE SÉCURITÉ NONETSENDING

Cette propriété interdit l'envoi des données sur le réseau. La figure 4.1 représente la chaîne des processeurs BeepBeep pour cette propriété.

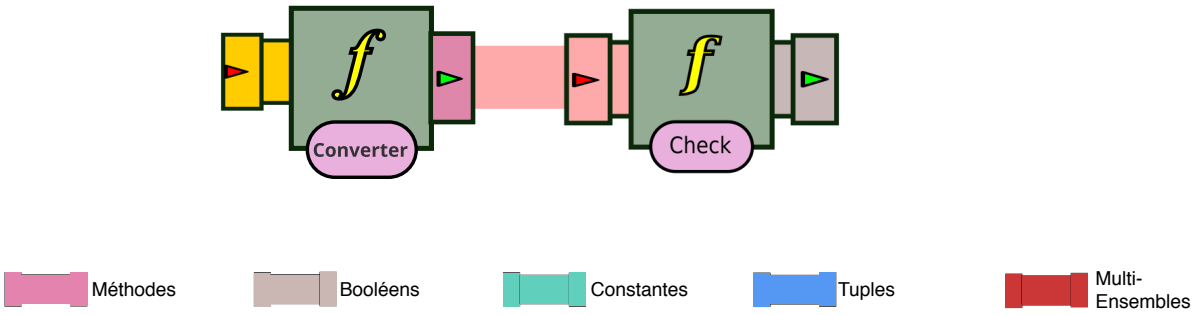


Figure 4.1 – Les processeurs BeepBeep pour la propriété de sécurité NoNetSending

La chaîne se compose de seulement deux processeurs : le premier extrait les trace ligne par ligne et les convertissent en événements. Ces événements sont acheminés au second processeur qui identifie selon leurs signatures les appels de méthodes qui envoient des données à travers le réseau. L'exécution est suspendue (via AspectJ) si la signature de la méthode correspond à une signature suspecte d'enfreindre la propriété.

4.2 PROPRIÉTÉ DE SÉCURITÉ LIMITBYTESWRITTEN

Cette propriété définit une limite stricte au nombre d'octets écrits ou générés par le programme, et elle peut être généralisée pour imposer une limite stricte cumulative à n'importe quelle ressource. La figure 4.2 donne la chaîne des processeurs BeepBeep pour la propriété LimitBytesWritten.

Cette chaîne se compose de sept processeurs : le premier extrait la trace ligne par ligne et convertie chaque ligne en événement. Le `Fork` (2) duplique la trace en deux flux, cela permet de filtrer les appels de méthodes qui effectuent une opération d'écriture (3,4) sur le système.

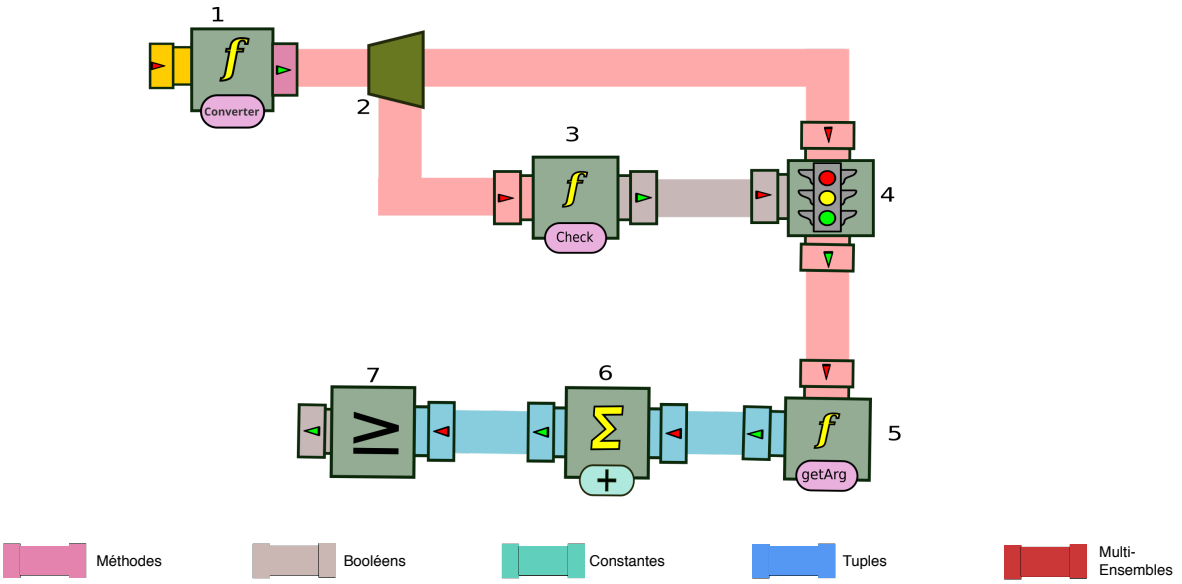


Figure 4.2 – Les processeurs BeepBeep pour la propriété de sécurité de sécurité LimitBytesWritten

Le processeur (3) a pour fonction de vérifier si la signature d'un appel de méthode est présente dans un fichier de signature associé à ce dernier. Ce fichier contient toutes les signatures que nous désirons vérifier, dans notre cas les signatures susceptibles d'écrire dans un fichier. Dans ce qui suit un aperçu du fichier en question :

```
java.io.OutputStream.write String
OtherClass.b String String
java.io.Writer.write java.lang.String
javax.swing.JTextArea.write java.io.FileWriter
```

Le processeur (4) de type `Filter` reçoit à son entrée des couples événement / booléen (E, B), E peut être soit de type appel de méthode soit retour de méthode. B est un booléen qui indique le résultat du test effectué par le processeur (3). Si $B = true$, (4) transmet l'événement à sa sortie, dans le cas contraire E est abandonné. Après être filtrés, les événements sont transmis au processeur (5) qui à pour fonction d'extraire le nombre d'octets écrits par chaque événement. Ce nombre est transmis au processeur suivant `Cumulative`. Ce processeur calcule la somme

des valeurs qu'il reçoit en entrée et envoie cette somme à sa sortie, par exemple pour la séquence de nombres $\{0, 8, 3, 11\}$, (6) produit à sa sortie la séquence $\{0, 8, 11, 22\}$. Finalement (7) vérifie si cette somme est supérieure à la limite fixée et envoie le verdict à l'utilisateur.

De même, en utilisant la même chaîne de processeurs, on peut représenter les propriétés de sécurité suivantes : LimitBytesRead, LimitCreatedFiles, LimitObservedFile et NoExec, NoJavaClassLoader, NoNetReceiveing, NoPrinting, NoReadingFiles, NoListingFiles respectivement et plusieurs autres propriétés de sécurité du projet Naccio.

La seule composante qui change pour ces propriétés est le fichier de signatures, en effet on change que les signatures des méthodes observées par le moniteur.

4.3 PROPRIÉTÉ DE SÉCURITÉ NOSENDAFTERREADING

La propriété de sécurité NoSendAfterReading (Schneider, 2000) propriété indique qu'après avoir lu à partir d'un fichier protégé, le programme n'est plus autorisé à accéder au réseau.

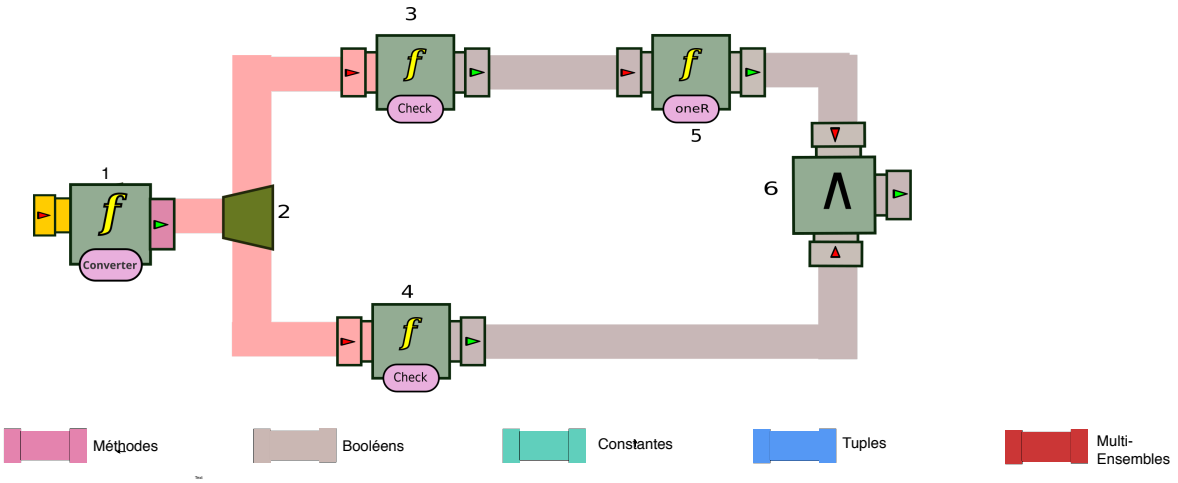


Figure 4.3 – Les processeurs BeepBeep pour la propriété de sécurité NoSendAfterReading

Pour cette propriété on a utilisé une chaîne de six processeurs (figure 4.3). Les lignes de la trace sont extraites et converties en événements (1). Le processeur (2) duplique le flux en deux. Les processeurs (3,4) vérifient respectivement si les appels de méthodes effectuent une opération de lecture ou d'envoi au sein du système. Le processeur (5) renvoie *True* en continu s'il détecte que le programme fait au moins une fois une opération d'écriture. Finalement, le dernier processeur (6) calcule la conjonction entre les couples de flux reçus de type booléen / booléen (A,B). Par exemple, pour $A = True, B = True$ (6) retourne $True$ ce qui signifie que la propriété a été enfreinte et cela sera signalé, dans les autres cas (6) retourne $False$.

4.4 PROPRIÉTÉ DE SÉCURITÉ ISAKEY

La propriété *« a is a key »* fait pas partie des propriétés Naccio, elle indique qu'une information donnée fournie dans la trace doit être unique et doit pas prendre la même valeur deux fois. Cette information peut consister en le paramètre d'une méthode ou sa valeur de retour.

Cette propriété peut être énoncée en utilisant des processeurs qui stockent dans une liste les valeurs apparues jusqu'ici dans l'exécution, et consultent cette liste avant de permettre l'exécution de l'appel de méthode suivant, la figure 4.4 représente les chaines de processeurs qui permettent de détecter l'enfreinte a cette propriété.

Le processeur (2) sert à extraire les événements de type Key, ce qui signifie que la valeur d'un paramètre de cet appel de méthode doit être unique. On extrait le champ qui correspond à la clef dans l'événement (3) et on vérifie si elle est déjà présente dans une table de hachage où sont stockés tous les couples clefs/méthodes (5). Si la clef correspond à une valeur déjà présente, cela signifie que la propriété a été enfreinte et cela sera signalé à l'utilisateur. Dans le cas contraire, la clef est stockée. Les résultats des processeurs (5) et (6) sont joints dans un tuple (8), par exemple si (8) reçoit à son entrée (true ,false) il les joint de la même forme ce

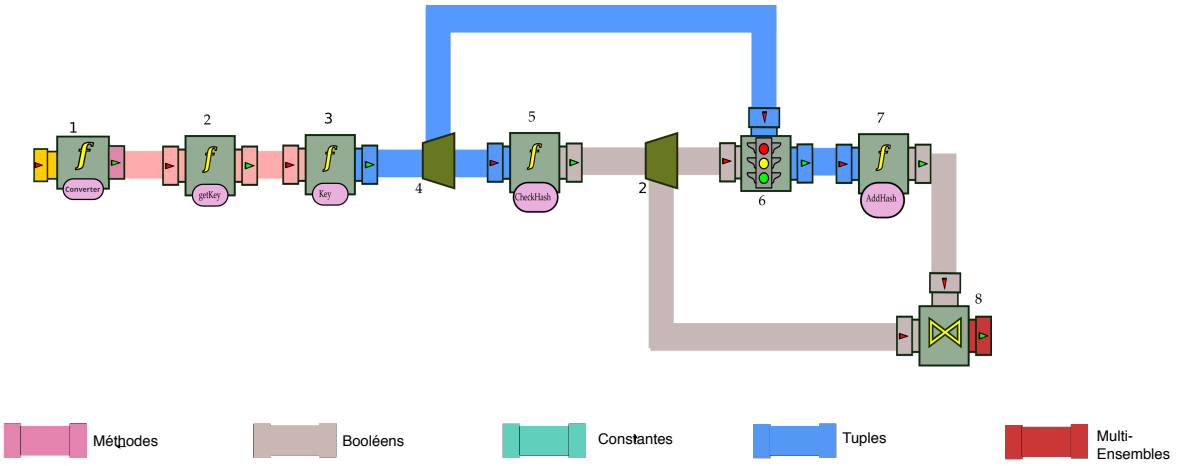


Figure 4.4 – Les processeurs BeepBeep pour la propriété de sécurité IsAKey

qui signifie que la clef n'est pas présente et qu'elle a été stockée.

4.5 PROPRIÉTÉ DE SÉCURITÉ ASREADASWRITE

La propriété *« as read as write »*, une variante de la propriété LimitBytesWritten dans laquelle le nombre d'octets pouvant être écrits est limité à la quantité de données qui a été lue jusqu'au moment présent au cours de l'exécution, plutôt que par un seuil.

Le flux original des événements de méthode est d'abord dupliqué en quatre copies (1), deux de ces copies sont données comme entrée aux processeurs (3,4) afin de filtrer les appels de méthodes qui écrivent au sein du système, de même pour les appels de méthodes qui lisent du système avec (2,5) respectivement. les processeurs (3,4) ont la même fonction, sauf qu'ils verifient pas les même signatures. On extrait et cumule la quantité de données écrites, ou lues avec (6,7,8,9), et au final on compare ces deux valeurs (10). Le verdict en sortie spécifie si le système enfreint la propriété de sécurité ou pas.

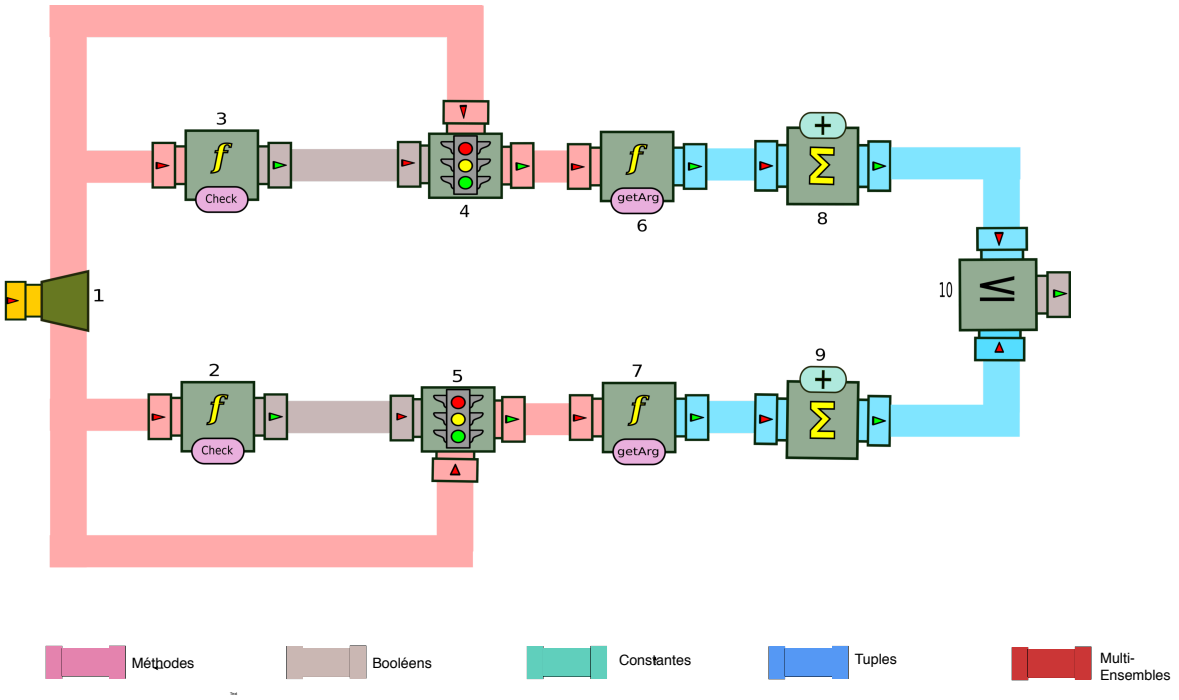


Figure 4.5 – Les processeurs BeepBeep pour la propriété de sécurité AsReadAsWrite

4.6 PROPRIÉTÉ DE SÉCURITÉ SAFELOCK

Cette propriété garantit que l'utilisation des primitives `acquire()` et `release()` sur les objets simultanés est effectuée de manière sûre pour les threads, le nombre d'appels de `acquire()` dans une méthode donnée doit être équilibré avec le nombre d'appels à `release()` (Hallé et al., 2014).

Le flux original des événements de méthode est d'abord divisé en deux (2), une copie est donnée comme entrée aux processeurs (3) pour vérifier les appels de la méthode `acquire`, de même pour les appels de la méthode `release` avec le processeur (4). Par la suite (5) et (6) prennent en entrée une valeur booléenne et a chaque réception de la valeur `True` incrémentent leurs compteurs et transmettent la valeur en sortie. Ces deux valeurs sont comparées dans le processeur (7) dont la sortie représente le verdict de la chaîne spécifiant si la propriété a été

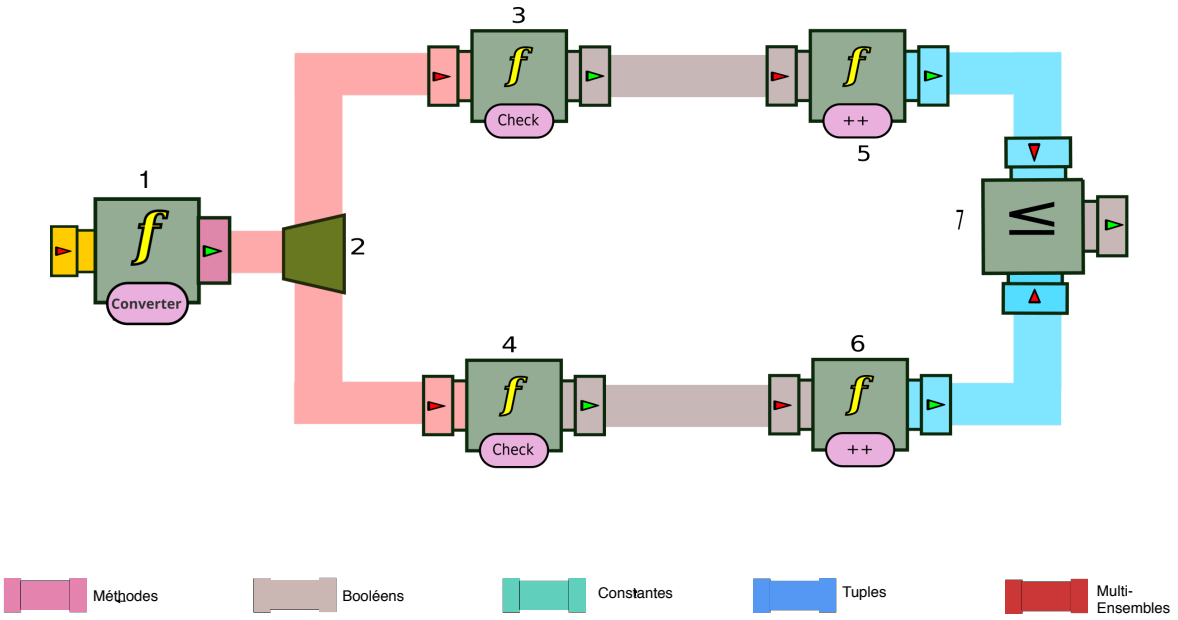


Figure 4.6 – Les processeurs BeepBeep pour la propriété de sécurité Safelock

respectée ou pas.

4.7 PROPRIÉTÉ CALLSEQUENCEPROFILING

Tel apparu dans (Boussaha et al., 2017), la propriété *« call sequence profiling »* spécifie, pour chaque méthode dans la trace, le nombre de fois où elle appelle directement chacune des autres méthodes. Cette information est fournie sous la forme d'un graphe orienté avec poids, dans lequel chaque sommet est étiqueté avec un nom de méthode, et un sommet de poids c est présent entre les sommets v_1 et v_2 si la méthode v_1 appelle la méthode v_2 c fois dans la trace (Boussaha et al., 2017).

Les graphes d'appel pourront être utilisés pour analyser l'apparition de nouvelles familles de programmes malveillants (Kinable et Kostakis, 2011). Nous donnons une représentation schématique de cette chaîne de processeurs dans la figure 4.7.

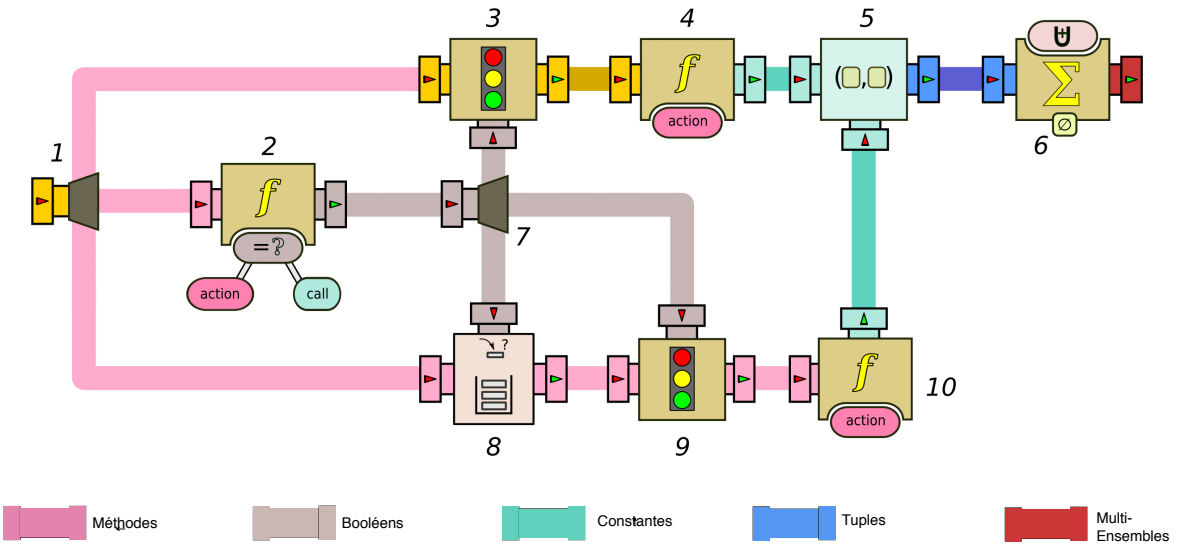


Figure 4.7 – Les processeurs BeepBeep pour la propriété de sécurité CallSequenceProfiling (Boussaha et al., 2017)

La figure 4.7 montre la chaîne de processeurs nécessaire pour calculer le graphe d'appel à partir d'une trace d'exécution. Le cœur de cette chaîne est le processeur `Stack`, représenté par le numéro 8 sur la figure. Il prend deux flux d'événements en entrée : le premier (côté gauche) est un flux d'événements de type arbitraire ; le second (côté supérieur) est un flux de valeurs booléennes. Ce processeur maintient en interne une pile d'événements reçus. À cette fin, le flux booléen agit comme un *push flag*. Lorsqu'un événement e et une valeur booléenne b arrivent aux entrées du processeur, deux situations peuvent se produire. Si $b = \top$, le sommet de la pile (s'il n'est pas vide) est sorti mais pas supprimé, et e est alors poussé sur la pile interne si $b = \top$. Si $b = \bot$, e est ignoré et l'élément en haut de la pile interne est retiré et rejeté.

Le flux original des événements de méthode est d'abord divisé en trois (1) ; l'une de ces copies est donnée comme entrée au processeur `Stack` (8), tandis qu'une autre est envoyée à un processeur `Function` (2). Ce processeur évalue la fonction qui compare le champ `action` de l'événement de méthode avec la constante `call` ; le résultat est un flux de valeurs booléennes,

indiquant si l'événement entrant est un appel de méthode ($\top$) ou un retour de méthode ($\bot$). Ce flux lui-même est divisé en trois (7), et l'une de ces copies est donnée comme drapeau de poussée du processeur `Stack`. La pile est donc appelée à pousser un événement entrant lorsqu'il s'agit d'un appel de méthode, et à faire apparaître le sommet de la pile lorsqu'il s'agit d'un retour de méthode. Par conséquent, la sortie du processeur `Stack` est l'événement de méthode correspondant à la méthode en cours dans l'exécution du programme.

Une troisième copie du flux d'événements d'origine est envoyée à un processeur `Filter` (3). Ce processeur reçoit deux entrées : un événement arbitraire e et une valeur booléenne b appelée *filtre*. L'événement e est produit si $b = \top$, sinon e est rejeté. L'indicateur de filtre, dans ce cas, est le résultat de l'application de la fonction `action` $=$ `call`, qui renvoie une valeur booléenne ; en d'autres termes, le processeur conserve uniquement les événements d'appel de méthode et filtre les retours de méthode. La même condition de filtrage est appliquée à la sortie du processeur `Stack` (9). Cependant, on remarquera que la sortie de 8 ne contient jamais de retour de méthode. C'est un exemple où l'indicateur est basé sur une condition qui n'est pas exprimée directement en termes d'événements qui sont filtrés.

Le résultat final de cette première partie de la chaîne est que les processeurs 3 et 9 produisent de manière synchrone des événements d'appel de méthode ; les événements aux positions correspondantes dans les flux représentent l'appelant d'une méthode (9) et la méthode étant appelée (3). À partir de ce moment, le reste du traitement est simple. Les deux événements sont traités de sorte que seule la valeur de leur champ `nom` est conservée (4, 10) ; ces deux valeurs sont ensuite jointes dans un tuple (5), et ces tuples sont ensuite accumulés dans un multiset en utilisant un `CumulativeProcessor` (6).

Le flux de sortie résultant de (6) est une séquence de multi-ensembles, chacun de la forme $\{(m_0, n_0), \ldots, (m_k, n_k)\}$; chaque tuple (m_i, n_i) est une paire appelant-appelé de noms de mé-

thodes. Le nombre de fois que chaque paire distincte se produit dans le multi-ensemble correspond au nombre de fois où n_i a été appelé à partir de m_i dans la trace. À partir de là, il est facile de prendre le multi-ensemble de tuples et de le transformer en un graphe orienté qui montre les dépendances pondérées entre les méthodes dans l'exécution observée.

Il est à noter que dans ce graphe entier, seule la fonction `action` (utilisée dans les processeurs 2, 4 et 10) et `Stack` processor (8) sont spécifiques à notre cas d'utilisation. Cela équivaut à moins de 150 lignes de code personnalisé. Tous les autres processeurs et fonctions sont génériques et sont déjà disponibles dans le noyau de BeepBeep ou dans l'une de ses palettes existantes.

4.8 PROPRIÉTÉ BYTEWRITTENGRAPH

La figure 4.8 montre une variante plus informative de la propriété LimitBytesWritten. Selon Boussaha et al. (2017) : « Le processeur BytesWrittenGraph fournit une liste de chaque méthode qui manipule chaque objet de données. Rappelons que les objets de données sont identifiés par référence dans la trace. Ce processeur, bien que n'appliquant pas les propriétés de sécurité, fournit des informations cruciales sur l'analyse de flux de données essentielle au débogage et à l'application des stratégies de flux de données. Il calcule le nombre d'octets écrits par chaque fonction qui le fait et exprime cette information sous la forme d'un graphe »

La chaîne du processeur commence par extraire de la trace les méthodes qui effectuent les opérations d'écriture, en ignorant toutes les autres lignes. Les paires contenant le nom de la méthode et le nombre d'octets écrits sont joints dans un tuple (6). Le processeur (7) divise son flux d'entrée en plusieurs flux distincts, chacun contenant des tuples provenant d'une seule méthode. Ceci permet d'agréger séparément le calcul du nombre d'octets écrits pour chaque méthode (processeurs 9 et 10). Le reste de la chaîne de processeurs regroupe ces informations

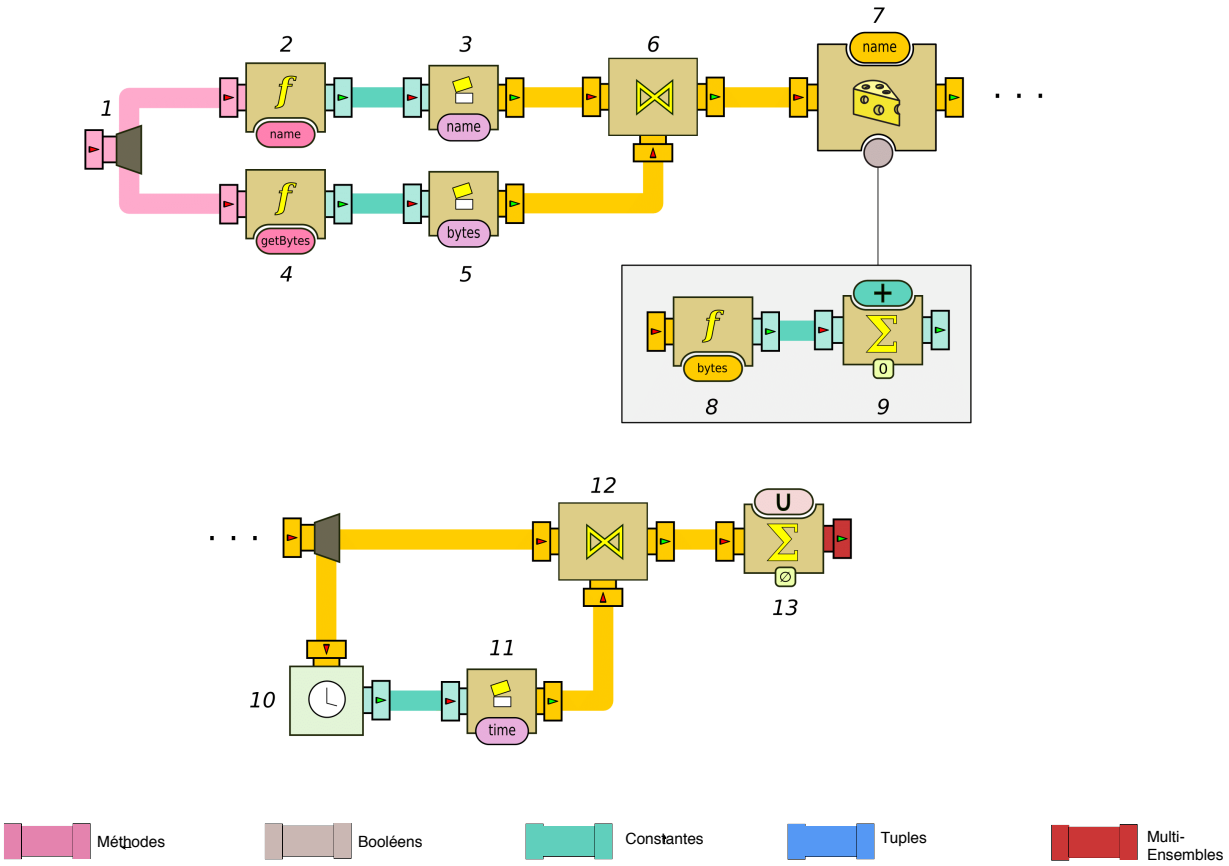


Figure 4.8 – Les processeurs BeepBeep pour la propriété de sécurité LimitBytesWrittenGraph (Boussaha et al., 2017)

avec un horodatage et met à jour une table de hachage en conséquence. Cette table de hachage peut alors servir de base à un graphe généré à la demande.

CHAPITRE 5

EXPÉRIMENTATION

Nous avons montré qu'une grande variété de propriétés de sécurité significatives ainsi que d'autres données d'intérêt liées à la bonne exécution de programmes tels que le profilage de séquence d'appel, peuvent être indiquées sous la forme de *processeurs*, dans le chapitre suivant (chapitre 5) pour l'expérimentation nous développons un outil qui s'appuie complètement sur notre approche.

Bien que les programmes que nous avons testés ne montrent aucun ralentissement lors des expérimentations, il faut aussi regarder l'impact de la technique en général. Nous considérons le temps de traitement du moniteur représentant l'impact de la technique sur l'exécution du programme et la facilité que l'utilisateur présente pour l'implémentation de moniteurs et pour étendre la liste des propriétés.

À cette fin, nous avons implémenté l'outil Stethoscope, un nouveau moniteur d'exécution utilisé pour appliquer les politiques de sécurité définies par l'utilisateur des programmes et des applets Java. Cet outil dont le code source est disponible ce `https://github.com/RacimBoussaha/Stethoscope`, s'appuie complètement sur l'approche proposée dans notre travail.

5.1 STETHOSCOPE

Nous avons montré précédemment qu'une multitude de propriétés de sécurité significatives ainsi que d'autres données d'intérêt liées à la bonne exécution de programmes, tels que *CallSequenceProfiling*, peuvent être décrits sous la forme de *processeurs*, qui représentent à leur tour une base pour le traitement des données en utilisant l'outil de traitement d'événement complexe BeepBeep. La seule difficulté qu'un utilisateur peut rencontrer réside dans la création de la chaîne de processeurs, qui doit être écrite en Java en utilisant la palette de processeurs disponibles dans BeepBeep.

Stethoscope permet aux utilisateurs de contourner le processus de conception de chaînes de processeurs entièrement. Au lieu de cela, l'utilisateur peut simplement instancier une chaîne de processeurs, à partir d'une liste restreinte mais croissante de modèles prédéfinis. BeepBeep (Hallé, 2016) sert de noyau de la structure de surveillance de l'exécution. Et nous comptons sur AspectJ (Kiczales et al., 2001) pour instrumenter et générer la trace d'entrée sur laquelle BeepBeep s'appuie pour effectuer l'analyse de l'application.

Le premier composant de d'outils est le traceur. La trace est générée par AspectJ d'une manière totalement transparente pour l'utilisateur, qui ne nécessite aucune connaissance particulière d'AspectJ pour manipuler Stethoscope.

Le deuxième composant de l'architecture est la chaîne de processeurs BeepBeep, qui définit la propriété de sécurité souhaitée.

5.1.1 EXECUTION

Cette section offre une démonstration de l'exécution de l'outil Stethoscope. Pour cela, il faut revenir à l'exemple de LimitBytesWritten, dont la chaîne de processeurs est présentée dans la

figure 4.2. Le code nécessaire pour implémenter cette chaîne est donnés dans le listing 5.1.

Listing 5.1 – Code source de la chaîne de processeurs pour la propriété LimitBytesWritten

```
1
2     LineReader feeder =
3         new LineReader(new FileInputStream(Trace));
4     Fork f1 = new Fork(2);
5     Fork f2 = new Fork(1);
6     Filter fil1 = new Filter();
7     FunctionProcessor converter =
8         new FunctionProcessor(StringToEvent.instance);
9     connect(feeder, converter);
10    connect(converter, f1);
11    connect(f1, 0, fil1, 0);
12    Function byte_count =
13        new ByteCount(
14            new Scanner(
15                new FileInputStream(Signature)));
16    Function max_fct = new SuperiorTo(max);
17    FunctionProcessor byte_count_p = new
18        FunctionProcessor(byte_count);
19    FunctionProcessor max_fct_processor = new
20        FunctionProcessor(max_fct);
21    connect(f1, 1, byte_count_p, 0);
22    CumulativeProcessor sum =
23        new CumulativeProcessor(
24            new CumulativeFunction<Number>(
25                Addition.instance));
26    connect(byte_count_p, sum);
27    connect(sum, f2);
28    connect(f2, max_fct_processor);
29    connect(max_fct_processor, 0, fil1, 1);
30    Pullable p = fil1.getPullableOutput();
```

L'utilisateur n'a qu'à sélectionner le programme Java ou le fichier JAR qu'il souhaite surveiller, l'interface représentée dans la figure 5.1 permet cela. L'application est ensuite démarrée depuis l'outil Stethoscope, comme montré dans la figure 5.2.

L'interface de l'outil permet à un utilisateur de spécifier une liste de noms de méthodes, et

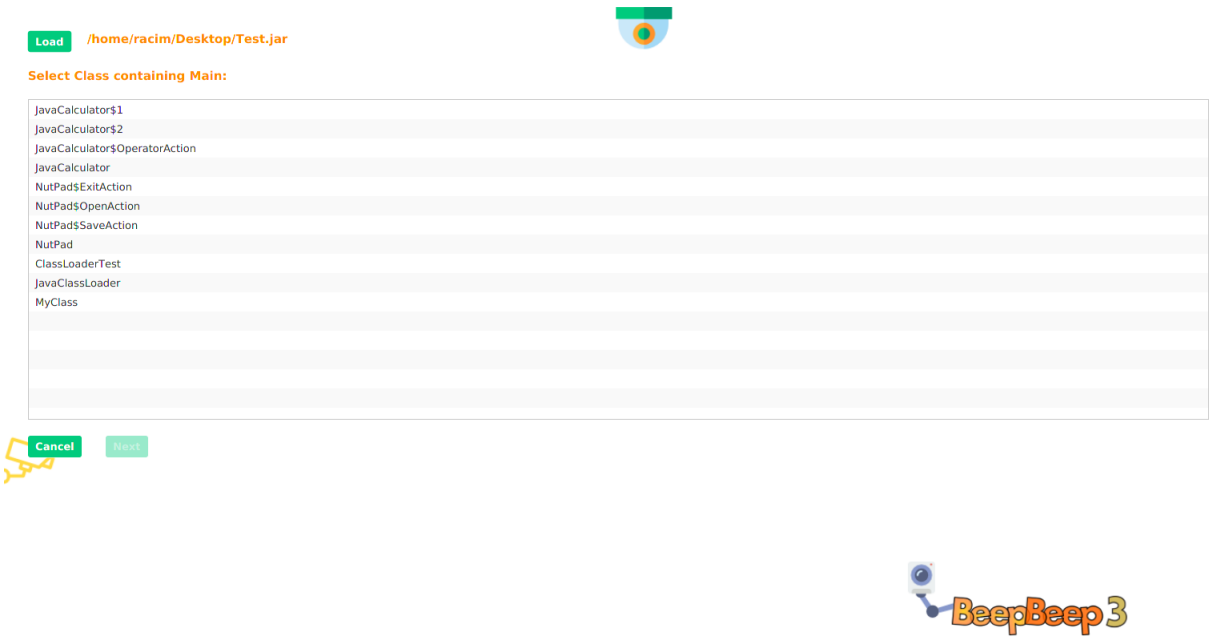


Figure 5.1 – L'interface pour la séléction du programme à exécuter

pour chacune de ces méthodes, un paramètre dont la valeur est ajoutée à un compteur. L'outil annule alors l'exécution si le compteur dépasse une valeur prédéfinie. L'utilisateur n'a qu'à spécifier cette valeur, ainsi que la liste des méthodes et paramètres d'intérêt.

L'interface de Sthetoscope est représentée dans la figure 5.3. Le haut de l'écran affiche la trace d'exécution en cours. En-dessous, l'utilisateur peut définir le nombre maximal d'octets que le programme cible est autorisé à écrire. Enfin, dans la partie inférieure de l'écran, l'utilisateur peut répertorier les méthodes pour lesquelles la propriété doit être appliquée. L'entier à droite du nom de la méthode indique l'index du paramètre dont la valeur doit être récupérée. Par exemple, la deuxième ligne entrée dans l'interface indique que la valeur du premier paramètre (numéro de paramètre 0) de tout appel à la méthode `Java.io.Writer.write java.lang.String` doit être ajoutée au total cumulé. Le résultat de l'analyse avec le verdict est montré dans la figure 5.4.

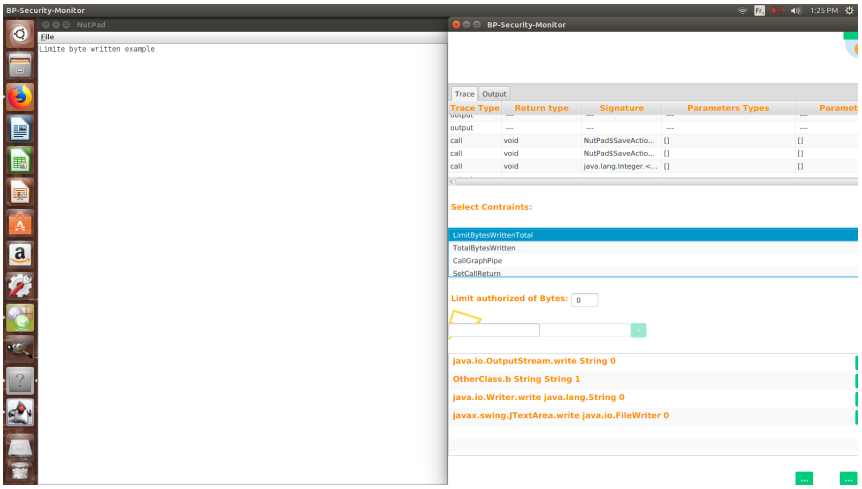


Figure 5.2 – L'interface pour l'exécution

Au moyen de cette interface, l'utilisateur n'a pas besoin de connaissances particulières de BeepBeep pour utiliser le moniteur. Tant que les propriétés qu'il souhaite appliquer sont des instanciations de celles présentes dans la palette de Stéthoscope, l'utilisateur n'a besoin que de paramétrer les chaînes de processeurs déjà présentes avec les noms de méthodes, les paramètres et toutes les autres valeurs requises par le processus de personnalisation.

Comme autre exemple, plusieurs propriétés présentes dans la bibliothèque de Naccio peuvent être implémentées simplement en listant une ou plusieurs méthodes que le programme ne peut pas appeler, quels que soient les paramètres de la méthode. C'est le cas de NoExecution, NoClassLoader, NoListing, NoReceive, NoSending, NoNetwork, NoPrinting et NoObserveTime. Chacune de ces propriétés est appliquée par notre outil en faisant appel à la même chaîne de processeurs, qui est paramétrée par une liste de méthodes d'intérêt.

Puisque l'utilisateur est libre de spécifier les méthodes dont il souhaite restreindre l'utilisation, cet outil permet en fait d'avoir une latitude dans l'élaboration de la politique qui n'est pas disponible ailleurs. En fait, l'utilisateur peut même choisir de restreindre l'utilisation des méthodes natives du programme, plutôt que celles de l'API Java.

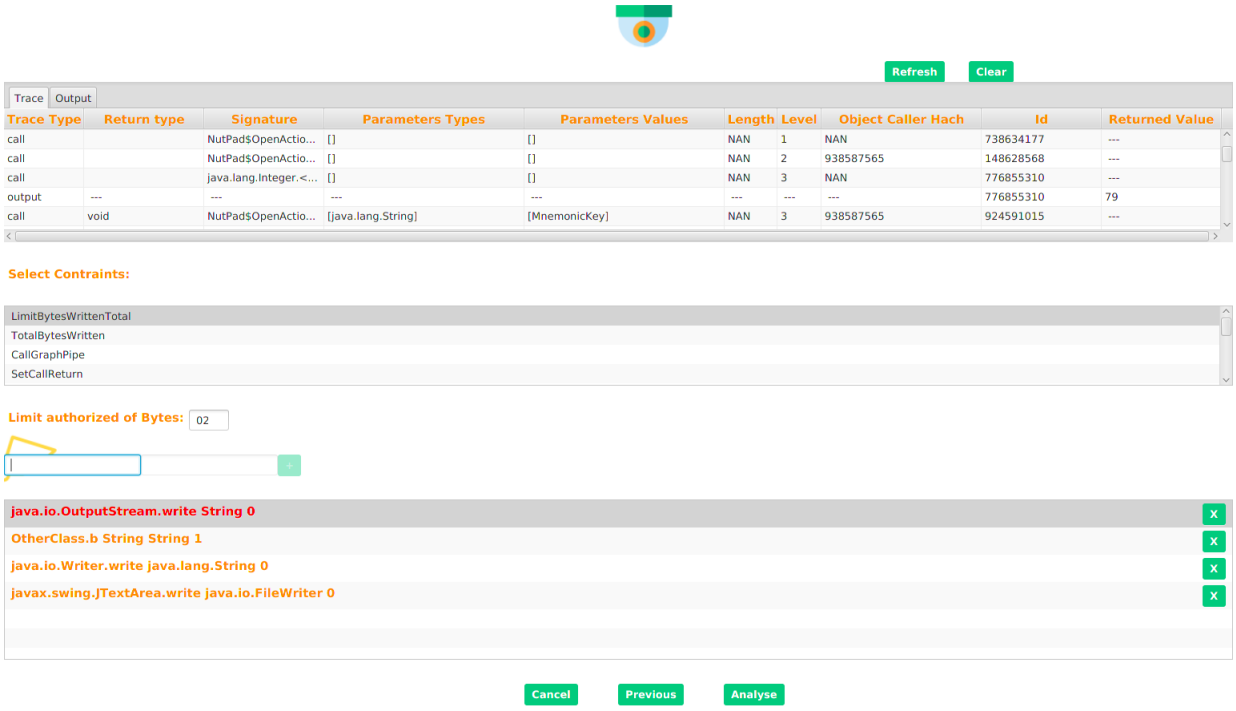


Figure 5.3 – L'interface pour la propriété LimitsBytesWritten

Le processus de surveillance utilisant BeepBeep est indépendant du mécanisme utilisé pour générer les traces, et alors qu'AspectJ présente également certaines capacités à fonctionner comme mécanisme d'application des politiques de sécurité, ce n'est pas nécessairement le cas pour les autres traceurs. L'utilisation de BeepBeep permet au mécanisme d'application de la sécurité d'être indépendant du traceur.

5.2 RÉSULTATS D'EXPÉRIMENTATION

Les spécifications de la machine dont nous nous sommes servi pour les tests sont données dans le Tableau 5.1.

Dans l'évaluation de l'impact de notre approche à la fois sur les programmes cibles et sur les utilisateurs du moniteur nous prenons en compte les facteurs suivants.

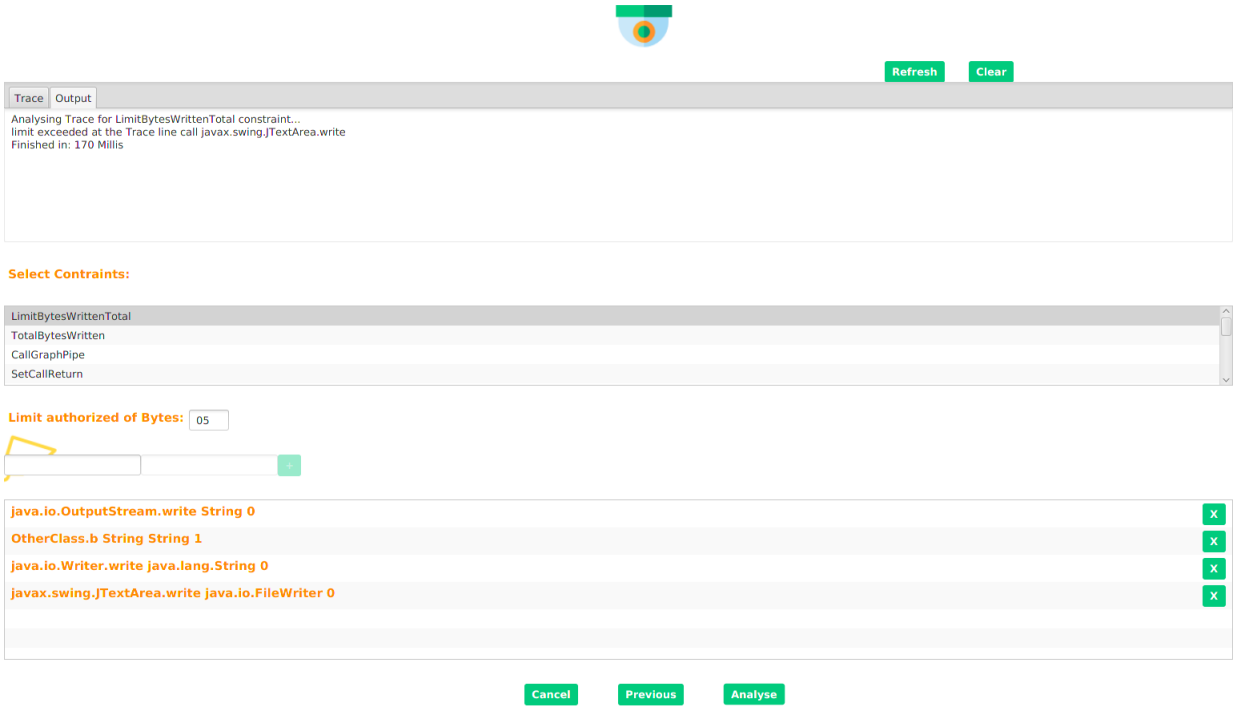


Figure 5.4 – Résultat de l'exécution du moniteur

Mémoire	11.6 GiB
Processeur	Intel Core i5-3320M CPU 2.60 GHz Quad Core
Graphique	Intel Ivybridge Mobile
Type OS	64-bit

Tableau 5.1 – Spécifications de l'ordinateur qui a servi aux tests

5.2.1 SÉCURITÉ

Tout système de sécurité doit respecter des exigences de sécurité. Le nôtre doit réussir à détecter les enfreintes des propriétés de sécurité pendant l'exécution du programme cible. La croissance rapide des « Malware » rend cette tâche de plus en plus difficile.

Comme tout autre système de sécurité basé sur la signature le nôtre dépend des signatures des appels de méthodes. Si un appel de méthode est susectible de provoquer une violation à une propriété de sécurité et qu'il n'est pas inclus dans liste des signatures associées à cette

propriété, la violation ne sera pas détectée et le système sous surveillance sera menacé. En conséquence, notre approche facilite la tâche d'ajout et de modification des processeurs et permet l'évolution de la sécurité du code. L'utilisateur peut ajouter des signatures qui peut potentiellement enfreindre une propriété quelconque en analysant la trace au fur et à mesure qu'elle est générée.

5.2.2 FLEXIBILITÉ

Dans la plupart des systèmes de sécurité on ne laisse pas à l'utilisateur la liberté de spécifier le niveau de sécurité qu'il accorde à un programme. En effet, ce programme doit respecter toutes les propriétés de sécurité disponibles dans ce système. Naturellement, analyser l'ensemble complet des propriétés de sécurité d'une exécution est plus couteux en temps et en performances que l'analyse d'un sous ensemble de celui-ci.

Notre système ajoute de la flexibilité et de la liberté à l'utilisateur dans son analyse. Il lui offre la possibilité de choisir les propriétés de sécurité qu'il désire être prises en compte pendant l'analyse et cela au moment de l'exécution. Selon le niveau de confiance qu'accorde l'utilisateur au programme sous *monitoring*, les propriétés adéquates sont sélectionnées.

De la même façon, cette méthode permet de tester un programme pendant la phase du développement de ce dernier ; elle donne la possibilité de vérifier si le programme respecte des propriétés de sécurité spécifiques.

5.2.3 EFFICACITÉ

Le travail du moniteur consiste à ramasser les détails de l'exécution du programme cible, sous la forme d'une trace d'exécution, de regarder si cette trace respecte les contraintes et de fournir un verdict ainsi que des détails sur l'exécution.

Pour évaluer l'impact que provoque notre méthode sur l'exécution du programme cible nous avons commencé par le tester sur une trace de 1 000 000 lignes. Ces traces ont été générées en appliquant la méthode décrite dans le chapitre 3.2 sur un programme de jeu Tetris implémenté en Java téléchargé à partir d'internet.

L'Overhead provoqué par le moniteur à l'exécution du programme monitoré est ensuite évalué. À cette fin, nous avons exécuté le moniteur en parallèle avec le programme et mesuré la différence du temps d'exécution avec celui du programme exécuté sans le moniteur. Nous avons fait le test pour quatre chaînes de processeurs de propriétés de différentes complexités, à savoir `NoSend`, `LimitBytesWritten`, `CallGraph` et `SafeLock`. Des détails sur l'implémentation de chacune de ces propriétés sont montrés dans 5.3.

Proriété	Overhead (ms)
CallGraphPipe	11797
SafeLock	5059
LimitBytesWritten	4795
NoSendOverNetwork	4341

Tableau 5.2 – Overhead pour des moniteurs testés sur une trace de 1 million de lignes

En deuxième lieu, nous avons coupé la trace en differentes longueurs allant de 0 à 1 000 000 en paliers de 100 000 événements. Le graphe représenté dans figure 5.5 montre le rapport entre le temps d'exécution (en millièmes de seconde) et la longueur de la trace (en nombre d'évènements ou lignes).

Comme on peut le voir dans ces résultats, les Overhead sont largement proportionnels au nombre de processeurs dans chaque chaîne de processeurs (Tableau 5.3). Comme la plupart des propriétés de sécurité ne nécessitent qu'un séquencement linéaire des processeurs, leur fonctionnement peut facilement être simplifié en fusionnant les opérations de plusieurs moniteurs dans une même classe. La seule chaîne de processeurs dont le temps d'exécution n'est

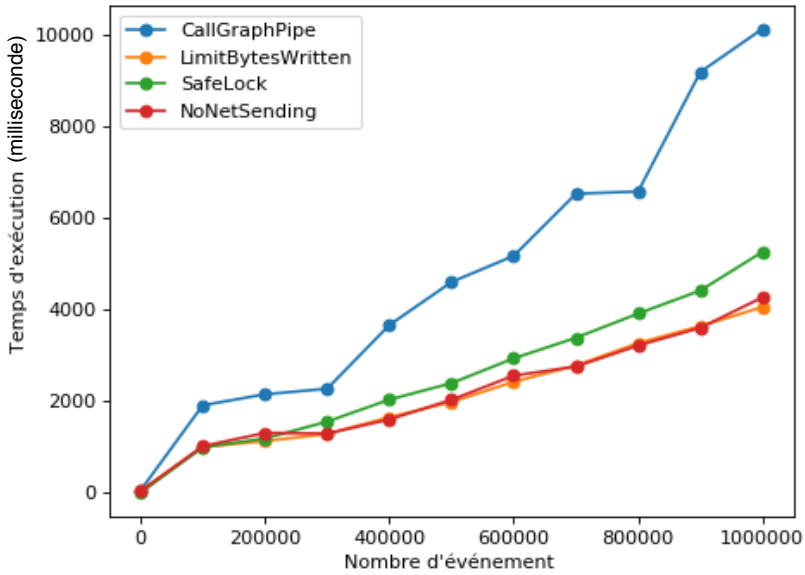


Figure 5.5 – Temps d'exécution pour plusieurs propriétés sur des traces de zéro à un million d'événements

pas négligeable est `CallGraph`, qui est destiné à être utilisé comme outil de développement.

5.2.4 FACILITÉ D'UTILISATION

Un avantage de notre approche est la facilité avec laquelle les propriétés de sécurité souhaitées peuvent être énoncées. Chaque processeur BeepBeep comprend en moyenne 20 lignes de code Java, contenues dans une seule classe. Les utilisateurs peuvent réutiliser ces blocs élémentaires en les enchaînant pour composer facilement des politiques complexes. Le tableau 5.3 détaille le nombre de processeurs, le nombre de lignes de code personnalisées (sans compter le code déjà présent dans la bibliothèque de modèles de BeepBeep) et le temps d'exécution des processeurs mentionnés dans ce travail.

Ce tableau illustre la facilité avec laquelle les processeurs BeepBeep peuvent être composées, nécessitant souvent seulement une quantité minimale de code personnalisé. Une fois ces

Proriété	Nb de processeurs	Nb lignes de code
CallGraphPipe	10	30
SafeLock	7	20
LimitBytesWritten	7	18
NoSendOverNetwork	2	6

Tableau 5.3 – Description des moniteurs testés

chaînes de processeurs implémentées, elles peuvent à leur tour être incluses en tant que composants de chaînes de processeurs pour des propriétés plus complexes avec l'ajout d'une seule ligne de code.

5.2.5 EXTENSIBILITÉ

De nature modulaire, BeepBeep a été conçu dès le départ pour être facilement extensible. Toute fonctionnalité au-delà des quelques processeurs intégrés présentés dans le chapitre 4 est implémenté par des processeurs personnalisés et des extensions de processeurs regroupés dans les palettes.

Comme on l'a vu, une palette est implémentée sous la forme d'un fichier .JAR chargé avec le programme principal de BeepBeep pour étendre ses fonctionnalités d'une manière particulière. Cette organisation modulaire est un moyen flexible et générique d'étendre le moteur à différents domaines d'application, de manière imprévue par ses concepteurs d'origine.

Au cours de notre travail nous avons implémenté quelques processeurs avec des fonctions atomiques personnalisées afin de mieux adapter BeepBeep au monde de la sécurité du code. Nous donnons ainsi un exemple de l'implémentation d'un de ces processeurs dans la figure 4.2. Le processeur `GetCalls` permet de filtrer les événements en ne laissant passer que ceux de type Call signifiant appel de méthode.

Listing 5.2 – Exemple de mise en œuvre d'un processeur BeepBeep personnalisé

```
1  public class GetCalls extends SingleProcessor {
2   public GetCalls() {
3    super(1, 1);
4   }
5   @Override
6   protected boolean compute(Object[] inputs, Queue < Object[] > outputs) {
7    Object o = inputs[0];
8    Object[] out = new Object[1];
9
10   if (o instanceof MethodCall) {
11    out[0] = (MethodCall) o;
12   } else {
13    return true;
14   }
15   outputs.add(out);
16   return true;
17  }
```

Ce fragment de code démontre la simplicité avec laquelle l'ajout de nouvelles fonctionnalités au moniteur est possible. Cela est réalisé simplement en étendant la classe `SingleProcessor` et en redéfinissant la méthode `compute`. La taille du code du processeur dépend de la complexité de la tâche qui y est associée. Ce processeur pourra être réutilisé dans le futur pour le développement d'autres propriétés de sécurité ; ceci est possible en instanciant des objets de la classe `GetCalls`.

CONCLUSION

Le code mobile est apparu comme une solution efficace aux défis de l'informatique dans les systèmes distribués. Néanmoins, les problèmes de sécurité restent omniprésents et peuvent constituer un frein à l'adoption de cette technologie, en partie parce qu'il est souvent difficile pour chaque utilisateur d'adapter la politique de sécurité qui régit son système à ses propres besoins.

Dans ce travail, il a été démontré comment BeepBeep, un processeur d'événements complexes, peut être utilisé comme un moniteur d'exécution pour appliquer un large éventail de propriétés de sécurité définies par l'utilisateur. Cette étude a servi également à illustrer les capacités de BeepBeep en tant qu'analyseur de traces. BeepBeep prend en entrée un flux de données, dans ce cas une trace d'exécution capturant les appels de méthode et les valeurs des paramètres. Cette manoeuvre est appelée instrumentation. Un système de monitoring doit être instrumenté, l'instrumentation sonde le système en exécution et rapporte les informations et les données sur l'exécution du système surveillé.

Pour le traçage et la collecte des données, le plugin de programmation orientée aspect destiné

au langage de programmation Java AspectJ a été utilisé. Ce plugin a permis l'implémentation d'un traceur qui s'occupe de l'instrumentation de notre moniteur BeepBeep en générant une trace de l'exécution sous forme de fichier texte qui sert d'entrée à ce dernier. La programmation orientée aspect a ainsi permis d'anticiper les actions sur le programme sous surveillance en définissant des `Advice` qui exécutent un code avant que chaque appelle de méthodes soit effectué. Ce code comprend l'analyse de l'appel de méthode par rapport aux contraintes de sécurité sélectionnées par l'utilisateur et interrompt l'exécution si une menace à la sécurité du système est détectée.

BeepBeep a la capacité d'analyser efficacement cette trace au moment de l'execution pour déterminer si elle est conforme à une spécification définie par l'utilisateur. BeepBeep peut également générer des informations de diagnostic utiles sur le comportement d'exécution du programme, qui peut à son tour être utilisé pour d'autres analyses de sécurité ou débogage. Cependant, la surveillance utilisant BeepBeep est agnostique du mécanisme utilisé pour générer les traces, et alors qu'AspectJ présente également certaines capacités à fonctionner comme un mécanisme d'application des propriétés de sécurité, ce n'est pas nécessairement le cas pour d'autres traceurs. L'utilisation de BeepBeep permet au mécanisme de sécurité d'être indépendant du traceur.

Pour exprimer les traitements souhaités, BeepBeep décompose la tâche en opérations élémentaires formant des chaînes de processeurs. Ces chaînes produisent le calcul souhaité. Ce travail vise en une partie à exprimer les processus de détection des propriétés de sécurité en chaînes de processeurs, un ensemble de propriété de différentes complexités ont été sélectionnés et modélisé avec cette abstraction, ensuite ces chaînes de processeurs permettant de détecter ces propriétés ont été implémentées avec l'outil d'« event stream processing BeepBeep ».

Afin de valider notre approche, nous avons implémenté Stethoscope, un outil de détection des violations des propriétés de sécurité construit autour de l'outil de «runtime monitoring» BeepBeep. Le logiciel est livré avec un ensemble de modèles de propriétés de sécurité de base qui peuvent être personnalisés pour permettre l'application d'une grande variété de politiques de sécurité. Le processus de personnalisation est simple et ne nécessite aucune connaissance particulière de BeepBeep ou du traceur.

Comme les autres mécanismes de sécurité du code, notre approche est précise, en ce sens qu'elle ne rejette que les programmes qui violent les propriétés de sécurité, permettant ainsi l'exécution du programme en toute sécurité. L'exécution n'est pas stoppée jusqu'à ce qu'une violation soit sur le point de se produire, permettant ainsi l'exécution de la plus grande partie possible compte tenu de la propriété de sécurité sélectionnée.

Les principaux avantages de l'approche proposée par rapport aux autres outils de suivi est la flexibilité ; facilité d'utilisation, extensibilité. Contrairement à de nombreux autres moniteurs de sécurité, chaque utilisateur est libre d'élaborer les politiques de sécurité qui correspondent le mieux à ses besoins particuliers. L'outil offre une interface à travers laquelle les utilisateurs peuvent personnaliser leurs propres stratégies de sécurité à partir d'un ensemble de modèles disponibles. Si aucun des modèles fournis n'est adéquat, les utilisateurs peuvent également créer les leurs, en codant la stratégie souhaitée en tant que chaînes de processeurs BeepBeep et en les téléchargeant sur l'outil sans grande difficulté.Par contre, notre approche exige à l'utilisateur d'acquérir des connaissances de base sur le fonctionnement de BeepBeep. En effet, afin de pouvoir créer de nouvelles propriétés en mettant en place des chaînes de processeurs, l'utilisateur doit connaître les principes de fonctionnement de notre outil d'analyse.

Une piste de recherche supplémentaire que nous explorons est de tirer parti des capacités de BeepBeep pour nous permettre d'exprimer un verdict plus informatif qu'une simple indication booléenne de la violation de la propriété de sécurité. Par exemple, le moniteur pourrait fournir des indications sur les parties de la trace qui ont causé les violations, évaluer sa gravité et suggérer des propriétés plus faibles qui sont respectées. Cette information pourrait, à son tour, servir de base à une réaction plus corrective à une violation potentielle que de simplement abandonner l'exécution.

Une étude comparative des autres approches avec celle présentée dans ce mémoire est prévue. Cela permettra de mieux apprécier la contribution de ce dernier au domaine de la programmation sécurisée.

Nous prévoyons d'étendre l'éventail des propriétés de sécurité de notre bibliothèque tels que :

- Les valeurs de paramètre d'une fonction donnée augmentent / diminuent toujours dans les appels consécutifs. Cette propriété peut servir à garantir l'exactitude de la fonction récursive.
- Après sa création, un objet donné n'est pas modifié (intégrité des données).
- Aucun thread n'est gelé pendant plus de 100 millisecondes avant de reprendre son exécution (liberté de famine).
- Indique si deux méthodes spécifiques fonctionnent sur le même objet ou bien fournit une liste d'objets manipulés par ces deux méthodes.

Nous prévoyaons aussi l'implémentation du moniteur de propriétés de sécurité adoptant nôtre approche dans d'autres interfaces et systèmes. Cela permettra d'analyser des programmes écrits dans d'autres langages de programmation que Java.

BIBLIOGRAPHIE

Ahmed, F., H. Hameed, M. Z. Shafiq, et M. Farooq. 2009. « Using spatio-temporal information in API calls with machine learning algorithms for malware detection ». In *Proceedings of the 2nd ACM workshop on Security and artificial intelligence*, p. 55–62. ACM.

Arnold, M. et B. G. Ryder. 2001. « A framework for reducing the cost of instrumented code », *Acm Sigplan Notices*, vol. 36, no. 5, p. 168–179.

Baker, J. et W. Hsieh. 2002. « Runtime aspect weaving through metaprogramming ». In *Proceedings of the 1st international conference on Aspect-oriented software development*, p. 86–95. ACM.

Barwise, M. 2010. « What is an internet worm ? », *BBC. Retrieved December*, vol. 9, p. 2014.

Bergeron, J., M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, N. Tawbi, et al. 2001. « Static detection of malicious code in executable programs », *Int. J. of Req. Eng*, vol. 2001, no. 184-189, p. 79.

Boussaha, M. R., R. Khoury, et S. Hallé. 2017. « Monitoring of security properties using beepbeep ». In *International Symposium on Foundations and Practice of Security*, p. 160–169. Springer.

Boyapati, C., R. Lee, et M. Rinard. 2002. « A type system for preventing data races and deadlocks in java programs ». In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, p. 211–230.

Brand, M., C. Valli, et A. Woodward. 2011. « A threat to cyber resilience : A malware rebirthing botnet ».

Chess, B. et J. West. 2007. *Secure programming with static analysis*. Pearson Education.

Cover, T. M. et J. A. Thomas. 1991. « Elements of information theory ».

Evans, D. et A. Twyman. 1999. « Flexible policy-directed code safety ». In *Proceedings of the 1999 IEEE Symposium on Security and Privacy (Cat. No.99CB36344)*, p. 32–45.

Evans, D. E. 2000. « Policy-directed code safety ». Thèse de Doctorat, Massachusetts Institute of Technology.

Gay, R., J. Hu, et H. Mantel. 2014. « Cliseau : securing distributed java programs by cooperative dynamic enforcement ». In *International Conference on Information Systems Security*, p. 378–398. Springer.

Gazet, A. 2010. « Comparative analysis of various ransomware virii », *Journal in computer virology*, vol. 6, no. 1, p. 77–90.

Goyal, M. et A. Sharma. 2015. « Survey on different kinds of malware and their detection », *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 5, no. 3.

Guilfanov, I. 2001. An advanced interactive multi-processor disassembler. http ://www.datarescue.com.

Hallé, S. 2016. « When RV meets CEP ». In Falcone, Y. et C. Sánchez, éditeurs, *Runtime Verification*, p. 68–91, Cham. Springer International Publishing.

Hallé, S., S. Gaboury, et R. Khoury. 2016. « A glue language for event stream processing ». In *Big Data (Big Data), 2016 IEEE International Conference on*, p. 2384–2391. IEEE.

Hallé, S. 2015. Beepbeep. `https://liflab.github.io/beepbeep-3/`.

———. 2016. « When rv meets cep ». In *International Conference on Runtime Verification*, p. 68–91. Springer.

———. 2018. *Event stream processing with BeepBeep 3*. unpublished.

Hallé, S., J. Vallet, et R. Tremblay-Lessard. 2014. « On piggyback runtime monitoring of object-oriented programs », vol. 17.

Hamann, T. et H. Mantel. 2018. « Decentralized dynamic security enforcement for mobile applications with cliseaudroid ».

Hughes, J. et G. Cybenko. 2014. « Three tenets for secure cyber-physical system design and assessment ». In *Cyber Sensing 2014*. T. 9097, p. 90970A. International Society for Optics and Photonics.

Jakobsson, A., N. Kosmatov, et J. Signoles. 2016. « Fast as a shadow, expressive as a tree : Optimized memory monitoring for c », *Science of Computer Programming*, vol. 132, p. 226–246.

Johnson, R., J. Hoeller, K. Donald, C. Sampaleanu, R. Harrop, T. Risberg, A. Arendsen, D. Davison, D. Kopylenko, M. Pollack, et al. 2004. « The spring framework–reference documentation », *Interface*, vol. 21, p. 27.

Khan, I. 2012. « An introduction to computer viruses : problems and solutions », *Library Hi Tech News*, vol. 29, no. 7, p. 8–12.

Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, et W. G. Griswold. 2001. *An Overview of AspectJ*, p. 327–354. Springer Berlin Heidelberg.

Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, et J. Irwin. 1997. « Aspect-oriented programming ». In *European conference on object-oriented programming*. Springer.

Kinable, J. et O. Kostakis. 2011. « Malware classification based on call graph clustering », *Journal in computer virology*, vol. 7, no. 4, p. 233–245.

Kirchner, F., N. Kosmatov, V. Prevosto, J. Signoles, et B. Yakobowski. 2015. « Frama-c : A software analysis perspective », *Formal Aspects of Computing*, vol. 27, no. 3, p. 573–609.

Kirda, E., C. Kruegel, G. Banks, G. Vigna, et R. Kemmerer. 2006. « Behavior-based spyware detection. ». In *Usenix Security Symposium*, p. 694.

Lam, L.-c., Y. Yu, et T.-c. Chiueh. 2006. « Secure mobile code execution service. ». In *LISA*, p. 53–62.

Leucker, M. et C. Schallhart. 2009. « A brief account of runtime verification », *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, p. 293–303.

Long, F., D. Mohindra, R. C. Seacord, D. F. Sutherland, et D. Svoboda. 2013. *Java Coding Guidelines : 75 Recommendations for Reliable and Secure Programs*. Addison-Wesley Professional, 1st édition.

Malin, C. H., E. Casey, et J. M. Aquilina. 2008. *Malware forensics : investigating and analyzing malicious code*. Syngress.

Mathur, K. et S. Hiranwal. 2013. « A survey on techniques in detection and analyzing malware executables », *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 3, no. 4.

McAfeeLabs. 2017. Mcafee labs threat report. `https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-dec-2017.pdf`.

Meetei, M. Z., A. Goel, et S. K. Wasan. 2011. « Observability using aspect-oriented programming for oo software testing », *International Journal of System Assurance Engineering and Management*, vol. 2, no. 2, p. 85–96.

Metz, E., R. Lencevicius, et T. F. Gonzalez. 2005. « Performance data collection using a hybrid approach ». In *ACM SIGSOFT Software Engineering Notes*. T. 30, p. 126–135. ACM.

Moore, D. J., V. Paxson, S. Savage, C. Shannon, S. Staniford-Chen, et N. Weaver. 2003. « Inside the slammer worm », *IEEE Security & Privacy*, vol. 1, p. 33–39.

Morgan, S. 2017. 2018 cybersecurity market report. `https://cybersecurityventures.com/cybersecurity-market-report/`.

Nagarakatte, S., J. Zhao, M. M. Martin, et S. Zdancewic. 2010. « Cets : compiler enforced temporal safety for c ». In *ACM Sigplan Notices*. T. 45, p. 31–40. ACM.

Nusayr, A. et J. Cook. 2009. « Using aop for detailed runtime monitoring instrumentation ». In *Proceedings of the Seventh International Workshop on Dynamic Analysis*, p. 8–14. ACM.

Pauli, D. 2017. Just give up : 123456 is still the world's most popular password. `https://www.theregister.co.uk/2017/01/16/123456_is_still_the_worlds_most_popular_password/`.

Peasley, R. 1998. *Using Visual Basic 6*. Que Pub.

Ray, D. et J. Ligatti. 2015. « A theory of gray security policies ». In *European Symposium on Research in Computer Security*, p. 481–499. Springer.

Rouse, M. 2016. An it security strategy guide for cios. `https://searchsecurity.techtarget.com/definition/information-security-infosec`.

Rouse, M. 2017. Source code analysis tools. `https://www.lemagit.fr/definition/Malware`.

Schatten, A., S. Biffl, M. Demolsky, E. Gostischa-Franta, T. Östreicher, et D. Winkler. 2010. *Best Practice Software-Engineering : Eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, Methoden und Werkzeugen*. Springer-Verlag.

Schiller, J. I. 2002. « Strong security requirements for internet engineering task force standard protocols ».

Schneider, F. B. 2000. « Enforceable security policies », *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 1, p. 30–50.

Signoles, J. 2018. E-acsl : Executable ansi/iso c specification language. http ://frama-c.com/download/e-acsl/e-acsl.pdf.

Tian, D., D. Qi, L. Zhan, Y. Yin, C. Hu, et J. Xue. 2017. « A practical method to confine sensitive api invocations on commodity hardware ». In *International Conference on Network and System Security*, p. 145–159. Springer.

Tran, A., M. Smith, et J. Miller. 2008. « A hardware-assisted tool for fast, full code coverage analysis ». In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, p. 321–322. IEEE.

Vacca, J. R. 2012. *Computer and information security handbook*. Newnes, second édition.

Vardi, M. Y. et P. Wolper. 1986. « An automata-theoretic approach to automatic program verification ». In *Proceedings of the First Symposium on Logic in Computer Science*, p. 322–331. IEEE Computer Society.

Varvaressos, S. 2014. « Étude de faisabilite du runtime monitoring dans les jeux vidéo ». Mémoire de maîtrise, Université du Québec à Chicoutimi.

Viega, J. et G. R. McGraw. 2001. *Building secure software : how to avoid security problems the right way*. Pearson Education.

Yadron, D. 2014. « Symantec develops new attack on cyberhacking », *Wall Street Journal*.

Zakinthinos, A. 1996. « On the composition of security properties ». Thèse de doctorat, University of Toronto.